MASTERING XENIX ON THE IBM PC AT
MASTERING XENIX ON THE IBM PC AT

HARLEY HAHN
I affectionately dedicate this book to my late mother Effi Hahn and to my father and stepmother, Murray and Marilyn Hahn. I will always remember their encouragement and support.
WHY YOU NEED THIS BOOK

I wrote this book to make IBM Xenix accessible. My goal is to introduce you to the world of Xenix in a thoughtful, careful manner. By the time you have finished this book, you will have a sound understanding of how to use the Xenix operating system.

This book is both a teacher and a reference. Every chapter begins with a list of the new words that will be introduced, along with their definitions. No concept is used before it is explained, and the explanations are easy to understand. As well, every important point is illustrated by at least one example.

I wrote this book for an intelligent person. I will not talk down to you; nor will I assume that you were born knowing about computers.

Buy this book. I am on your side.

ACKNOWLEDGMENTS

By far, my favorite task with each book is to write the acknowledgments. More than most people suspect, every writer has a clutch of friends and advisors who provide the encouragement and assistance it takes to carry out a large project. I am pleased to be able to mention their names.

Most important, I thank Dianne MacLeod for her constant guidance, encouragement, research, and proofreading.

For expert advice on matters technical, I thank Dr. Barton P. Miller of the University of Wisconsin, Stewart Walton of Growth Systems (San Diego), and Gail Anderson. For friendly, competent help with hardware, I thank William Cook of Cook's Computer Electronics (San Diego).

For thoughtful reading of the manuscript, I thank Al Melkonian of Culler Scientific and Alan Simpson.

At IBM, I thank Rosemary Morrissey and Clovis Tondo, and at Microsoft, I thank Paul Mulholland.

And for important technical information on the correct spelling of "MXYZPTLK," I thank Robin Royal of the Comic Kingdom Graphic Fantasy Shop (San Diego).

Harley Hahn
# CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>How to Use This Book</td>
<td>ix</td>
<td></td>
</tr>
<tr>
<td>Chapter 1</td>
<td>INTRODUCTION TO XENIX</td>
<td>1</td>
</tr>
<tr>
<td>Chapter 2</td>
<td>THE KEYBOARD</td>
<td>10</td>
</tr>
<tr>
<td>Chapter 3</td>
<td>STARTING TO USE XENIX</td>
<td>18</td>
</tr>
<tr>
<td>Chapter 4</td>
<td>LEARNING TO USE XENIX</td>
<td>30</td>
</tr>
<tr>
<td>Chapter 5</td>
<td>COMMANDS THAT YOU CAN USE RIGHT AWAY</td>
<td>39</td>
</tr>
<tr>
<td>Chapter 6</td>
<td>TWO USEFUL PROGRAMS</td>
<td>55</td>
</tr>
<tr>
<td>Chapter 7</td>
<td>UNDERSTANDING THE XENIX FILE SYSTEM</td>
<td>67</td>
</tr>
<tr>
<td>Chapter 8</td>
<td>USING THE XENIX FILE SYSTEM</td>
<td>83</td>
</tr>
<tr>
<td>Chapter 9</td>
<td>USING FILES AND DIRECTORIES</td>
<td>101</td>
</tr>
<tr>
<td>Chapter 10</td>
<td>THE vi EDITOR—THE BASICS</td>
<td>122</td>
</tr>
<tr>
<td>Chapter 11</td>
<td>THE vi EDITOR—DISPLAYING TEXT FILES AND MOVING THE CURSOR</td>
<td>134</td>
</tr>
<tr>
<td>Chapter 12</td>
<td>THE vi EDITOR—CREATING AND MODIFYING TEXT FILES</td>
<td>152</td>
</tr>
<tr>
<td>Chapter 13</td>
<td>The vi EDITOR—THE REST OF WHAT YOU NEED TO KNOW</td>
<td>169</td>
</tr>
<tr>
<td>Chapter 14</td>
<td>CONTROLLING AND COMBINING COMMANDS</td>
<td>199</td>
</tr>
<tr>
<td>Chapter 15</td>
<td>READING AND WRITING TO DEVICES</td>
<td>215</td>
</tr>
<tr>
<td>Chapter 16</td>
<td>SEARCHING, SORTING, AND COMPARING</td>
<td>231</td>
</tr>
<tr>
<td>Chapter</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>------------------</td>
<td>------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Chapter 17</td>
<td>SHARING AND RESTRICTING ACCESS TO FILES</td>
<td>259</td>
</tr>
<tr>
<td>Chapter 18</td>
<td>THE XENIX MAIL SYSTEM</td>
<td>266</td>
</tr>
<tr>
<td>Chapter 19</td>
<td>USING DISKETTES FOR BACKUPS AND WITH DOS</td>
<td>285</td>
</tr>
<tr>
<td>Chapter 20</td>
<td>PROGRAMMING THE SHELL—BEGINNING CONCEPTS</td>
<td>298</td>
</tr>
<tr>
<td>Chapter 21</td>
<td>PROGRAMMING THE SHELL—ADVANCED CONCEPTS</td>
<td>319</td>
</tr>
<tr>
<td>Appendix A</td>
<td>ASCII CONVERSION TABLE</td>
<td>339</td>
</tr>
<tr>
<td>Appendix B</td>
<td>SUMMARY OF vi COMMANDS</td>
<td>345</td>
</tr>
<tr>
<td></td>
<td>Glossary</td>
<td>354</td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>366</td>
</tr>
</tbody>
</table>
HOW TO USE THIS BOOK

If you have never used Xenix before, read through this book chapter by chapter. At the beginning of each chapter, you will see a list of the new words. Skim the list to get an idea of what you will learn. As you read, refer back to this list if you forget what a particular term means. For your convenience, all the new words are collected in a glossary at the back of the book.

This book will serve you best if you read near your computer. Make a habit of trying things out as you learn.

If you have used Xenix or a similar system before, start with Chapter 1 and skim through the book. Read those sections that explain concepts with which you are unfamiliar. I particularly suggest that you read Chapters 20 and 21 for important advice on how to design shell scripts. As well, you will find the chapters on the vi editor to be complete and filled with examples. If you have used the editor already, I direct your attention to the summary of commands in Appendix B.
CHAPTER 1

INTRODUCTION TO XENIX

INTRODUCTION
BEFORE YOU START
THE BASICS OF XENIX
INTRODUCTION

Most of what you can do with your computer requires a certain amount of background preparation. Part of this means understanding what you are working with and what it can do. In this chapter, I will introduce you to Xenix. If you have never used a computer before, you will start to learn the basic concepts and terms that you need.

New Words in This Chapter

OPERATING SYSTEM
a complex program that coordinates the overall functioning of a computer

UNIX
an operating system originally developed at Bell Labs

XENIX
an operating system based on UNIX; Xenix was developed by Microsoft Corporation for use with personal computers

USER
a person who is using the computer system

USERID
an internal Xenix identification name; Xenix knows each user by userid; pronounced: "user-eye-dee"

PASSWORD
a sequence of characters that a user must specify to access the Xenix system; each userid has its own password

PROGRAM
a set of instructions that can be carried out by a computer

EXECUTE [a program]
to perform the instructions in a program

RUN [a program]
same as EXECUTE

DATA
information used by a program

INPUT
information supplied to the program while it is executing
READ
  to take in information during the execution of a program

OUTPUT
  information generated by a program

WRITE [output]
  to generate information during the execution of a program

INPUT/OUTPUT
  the general concept of reading and writing information; abbreviated: I/O

DISPLAY
  the television-like device upon which the computer writes information;
  to write information upon a display

PRINT
  to write output on paper using a printer
  (Xenix manuals often use this term to refer to writing information on a display)

TERMINAL
  the keyboard and display unit with which you communicate with Xenix

TTY
  same as TERMINAL; pronounced: "tee-tee-why"

CONSOLE
  the name of the terminal consisting of the keyboard and display that is part of the computer on which Xenix resides

SYSTEM MANAGER
  the person who maintains the Xenix system

root
  a special userid that affords the user many extraordinary privileges;
  usually used only by the system manager

SUPERUSER
  a person using the root userid

What Is Xenix?

An OPERATING SYSTEM is a complex program that coordinates the overall functioning of a computer. XENIX is an operating system. It maintains files, transfers information between the parts of the computer,
manages the memory, and so on. Most important, Xenix carries out the
commands that you give it.

Xenix is a member of the UNIX family of operating systems. Unix is
the name of an operating system originally developed at Bell Labs.
Today, there are many members of the family designed for a variety of
collectors. Xenix was designed by the Microsoft Corporation for use
with personal computers. IBM offers a modified version of Microsoft
Xenix to use with IBM personal computers.

This book teaches you how to use IBM’s Xenix, officially called
"IBM Personal Computer XENIX." The Unix family is similar enough so
that once you learn IBM Xenix, you will be able to use any Unix-based
operating system.

Xenix is one of the most sophisticated operating systems for small
computers. There are many built-in facilities that are not found in other
systems. These include:

• a software development system
• a text processing system
• three different text editors
• an electronic mail system
• a network communications system

There are also many smaller tools for sorting, calculating, con­
verting between different systems of measurement, and on and on.
Overall, Xenix provides the environment in which you work.

BEFORE YOU START

What Is It Like, Working with Xenix

The term USER refers to a person who is using the computer. Each
user has a name by which Xenix knows him or her. This name is called
USERID (pronounced: “user-eye-dee”). There is a PASSWORD for every
userid. There might be many people registered with the system, each
having their own userids and passwords. Xenix allows several users to
use the computer at the same time.

To work with Xenix, you use a keyboard and display unit to
communicate with the computer. To start working, you connect this unit
to Xenix. Xenix will ask for your userid and password. Once Xenix has verified that you are a legitimate user, it will set up your working environment and wait for you to type a command.

The working session consists of your typing commands and Xenix carrying out those commands. The idea is to choose the commands that do the work you want.

When you are finished, you signal that you have no more commands to be carried out. Xenix will then end your work session. If you are working on your own personal computer, you can stop Xenix and turn off the computer.

A Few Basic Concepts

If you are new to computers, you need to know some basic terms. I will introduce most new terms as you need them. However, there are a few general ones that you should learn before you start. If you have used computers before, just skim this section.

A computer is a general-purpose machine that carries out instructions. A set of instructions is called a PROGRAM. When a computer carries out the instructions in a program, we say that the computer EXECUTES or RUNS the program.

Most programs process information. We refer to the information that a program processes as DATA. If the data is to be supplied while the program is executing, we call the data INPUT, and we say that the program READS the data. If the program generates data that is to be stored or displayed, we call the data OUTPUT, and we say that the program WRITES the data.

For example, say you run a program that processes information that you type at the keyboard. When the processing is finished, the program prints a table of updated information on the printer. Using computer terms, we would say that the program reads its input from the keyboard and writes its output to the printer.

The general concept of reading or writing data is referred to as INPUT/OUTPUT, often abbreviated as I/O. For example: Reading input is the only form of I/O that is possible using the keyboard. Writing output is the only form of I/O that is possible using the printer.

The word DISPLAY refers to the television-like screen upon which the computer writes information. The same term is often used as a verb to refer to writing output to the display screen. For example: If you make a mistake, the program will display an error message on the display.
In this book, I will use the word PRINT to refer to writing output on paper using a printer. However, Xenix reference manuals often use "print" when they mean "display." This is a carry-over from the days when Unix was developed. In those days, programmers used typewriter-like devices that printed data, rather than the newer television-like devices that display the data on a screen.

So, if you read in a manual that a certain command "prints" the names of all your files, you will understand that the command really displays the information. When such manuals really do refer to printing data, they will be explicit. For example, a manual might say that a certain command "writes its output to the printer."

The Xenix Philosophy

Before you start, you should understand something of the Xenix philosophy. This will help you work better in the Xenix environment.

The most important thing to understand is that Xenix is easy to use but not easy to master. Here is an analogy: A bicycle is much more difficult to master than a tricycle. However, once you can ride both, you find the bicycle more maneuverable, efficient, and generally easier to use. Similarly, it may take you longer to learn Xenix than it would another operating system, but once you do, you will find that it will carry you long distances quickly. One thing about Xenix: It is particularly well suited to human beings, once you are used to it.

I tell you this now so that when something doesn't work right and it takes you awhile to figure out what to do, you won't become frustrated. You are learning to use a powerful tool, and it may take some time.

One of the nice things about Xenix is that it makes sense. It has a logic about it that is almost always consistent. Once you become experienced, you will often be able to guess at how something works.

As a general rule, you should learn the concepts and not try to memorize the details. As you work, you will find that you will automatically begin to memorize the details of the parts of Xenix that you use the most. Until you do, it is perfectly fine to look up information in this book and in the reference manuals.

Here is an example: There are two commands that you can use to list the names of the files with which you are working. One command (ls) has 12 options, and the other (lc) has 22 options. It would be silly to try to memorize all the options. Rather you should learn what these commands do and when to use one or the other. When you need to use a particular option, you can look it up.
Another important point about Xenix is that it is designed as a collection of tools. Each command that you enter should be thought of as a request for a tool to perform a particular task. For example, the `ls` and `lc` commands that I just mentioned are really tools for listing the names of your files; the `date` command is a tool for displaying the time and date; and so on.

Rather than have one command perform many different tasks, Xenix was designed so that each command performs one task well. Imagine a carpenter's toolbox. A tool that could be a hammer and a screwdriver and a wrench all at the same time would be unwieldy. Better to have one tool for each task. The power of Xenix comes from the ease with which you can combine tools to suit the job at hand.

Thus, Xenix continually requires your creativity. Every time you have a task to be done you must ask yourself, Can it be done by one tool? If not, what combination of tools should I use? Once you become proficient in combining tools, you will take delight in using your ingenuity to solve problems.

Xenix allows you to combine the tools in many different ways. You will often find that there is more than one way to perform a task. A good plan is to learn one way to do each important task and learn others as the need arises.

One last point: Xenix is terse. The frequently used commands have short names to make them easy to use. For example, the commands to copy, remove, and move a file are `cp`, `rm`, and `mv`, respectively.

Similarly, Xenix displays messages only when necessary, and even then, the messages are short. There is no special message to tell you that a command has been successfully carried out. And when something goes wrong, Xenix will typically describe the situation in only a few choice words.

At first, the abbreviations and silence may seem cryptic and abrupt. After awhile, you will come to appreciate the brevity.

**THE BASICS OF XENIX**

**The Terminal**

The keyboard and display unit with which you communicate with Xenix is called a TERMINAL. Each terminal is connected to the computer in which Xenix resides. A terminal may be a device that is not
much more than a keyboard and a display, or it may be another computer.

(If you were to draw a diagram showing how the various parts of the computer are connected, each terminal would be at the end of a communication line. Thus, the name *terminal*.)

Sometimes the name TTY (pronounced: “tee-tee-why”) is used to refer to a terminal. For example, the Xenix command to set the characteristics of your terminal is called *stty*. The letters “tty” are an abbreviation for “teletype.” (A teletype is an outdated electromechanical device that was used as one of the first computer terminals.)

When you run Xenix on your own computer, your keyboard and display act as a terminal. This terminal is given a special name: the CONSOLE. If someone else wants to use your system, they must connect their terminal with your computer, either by a direct wire or over the telephone.

Xenix can handle up to 16 terminals at a time: the console, and 15 others. These others are referred to as TTY1, TTY2, and so on. In other words, up to 16 people at a time can use Xenix. However, the number of users is usually limited to three or four; otherwise, the system becomes too slow.

In practice, the number of terminals is limited by the number of communication lines that you can connect to your computer. With the IBM PC AT computer, communication lines connect to adapters that have outlets in the back of the computer. The PC AT can have only two such adapters, and IBM adapters have only one outlet each. If you want to connect more than two communication lines, you will have to buy non-IBM adapters that come with extra outlets.

**Users and Userids**

Xenix keeps a table of all the userids that are registered with the system. When you use Xenix, you must tell Xenix which userid you want to use. Of course, you must know the password for that userid.

Generally, each user has a single userid. The userid is often picked to be a short form of the person's name. For example, a person named Ron Dragushan might pick the userid *ron* or *drag*. The password is usually chosen to be something that cannot be guessed easily, such as “44%cxa”.

Make sure that you understand the difference between a user (a person) and a userid (an internal Xenix identification name). For example, it is possible for two people, each with a terminal, to use Xenix at the same time under the same userid.
The System Manager

There are two different types of users: the SYSTEM MANAGER and everybody else. As the name implies, the system manager is the person who maintains the entire system, the one who attends to the needs of Xenix and of the users. For example, it is the system manager who registers new users.

If you are running Xenix on your own personal computer, then you will be the system manager. In particular, you will have to install Xenix. To do this, follow the instructions in the Installation Guide that comes with the system. Before you can start working with Xenix, you will have to give yourself a userid and password.

The userid that the system manager uses to carry out official duties is always the same. It is root. (This name will make sense when you learn about the file system.) When the system manager installs Xenix, he or she chooses the password for the root userid.

The system manager always has a regular userid for personal use. It is only when performing system maintenance that the root userid is employed. This is because the root userid comes with many privileges that are otherwise unavailable. The system manager uses the root userid only when necessary, to avoid damaging the system accidentally. When the system manager is using Xenix under the root userid, he or she is referred to as the SUPERUSER.
CHAPTER 2

THE KEYBOARD

INTRODUCTION

THE THREE PARTS OF THE KEYBOARD

INPUT FROM THE KEYBOARD
INTRODUCTION

The main way that you communicate with Xenix is by using the keyboard. Although the keyboard resembles a typewriter, there are extra keys with special meanings that you must learn. In this chapter, I will introduce the various keys. As well, you will learn a little about how Xenix handles the information that you type.

New Words in This Chapter

FUNCTION-KEY AREA
the left-hand part of the keyboard, containing the ten keys labeled <F1> through <F10>

TYPEWRITER-KEY AREA
the center part of the keyboard, containing keys similar to those on a typewriter

NUMERIC KEYPAD
the right-hand part of the keyboard, containing a variety of special keys

space
the character produced by pressing the <SPACE> key

newline
the character produced by pressing the <ENTER> key

ENTER
to type information and then press the <ENTER> key

tab
the character produced by pressing the <TAB> key

CURSOR-CONTROL KEYS
the four keys with arrows on the numeric keypad that change the position of the cursor

ECHO
to display characters on the screen as you type them

BUFFER
a temporary holding area for data that is being transferred

KEYBOARD BUFFER
the buffer in which Xenix collects the characters that you type at the keyboard
CURSOR
the blinking underscore symbol that marks the current position on the display screen

Using the Keyboard with Xenix

In this chapter, I describe how to use the keyboard that is part of the IBM PC AT computer. When you use a PC AT to run Xenix, this is the keyboard that is associated with the console.

You can use Xenix with many different types of terminals. Each type will have its own keyboard design. However, all of the important keys should be somewhere on the keyboard. If you are using a different kind of keyboard, follow along. As I describe a key, find the corresponding key on your keyboard.

Whenever I refer to a key, I will enclose it in < and >. For example, <a> refers to the "a" key; <ENTER> refers to the "enter" key.

Xenix uses most of the keys of the keyboard in a straightforward manner. However, there are a few rules that you must follow, and there are a few keys that have special purposes.

The PC AT keyboard has three areas: the FUNCTION-KEY AREA (to the left), the TYPEWRITER-KEY AREA (in the middle), and the NUMERIC KEYPAD (on the right).

THE THREE PARTS OF THE KEYBOARD

The Function-Key Area

The function-key area contains ten keys, labeled <F1> through <F10>. These keys do not have any particular use in Xenix. You can set them to whatever you want. (Use the setkey command, explained in Chapter 5.)

The Typewriter-Key Area

The typewriter-key area works much the same as a typewriter. To type an upper-case (capital) letter, hold down one of the <SHIFT> keys and press the letter. The same goes for typing a special character that is
The Three Parts of the Keyboard

on the top half of a key. For example, to type "$", use <SHIFT 4>.

If you have to type many upper-case letters in a row, press <CapsLock>. All the letters will be upper case, until you press <CapsLock> again. When your press <CapsLock>, a small green light in the top right-hand corner of the keyboard comes on as a reminder. When you press <CapsLock> again, the light will go off. <CapsLock> only works for letters, not for special symbols.

Xenix distinguishes between upper and lower case. For example, to use the command that displays the date, you must type date, not DATE or Date. When you are typing Xenix commands, you will be using lower case almost all the time.

The <CTRL> key on the left-hand side of the typewriter-key area acts like a second type of shift key. You hold it down as you press another key. For example, if I want you to hold down the <CTRL> key and press <a>, I will write <CTRL a>.

Just as <SHIFT a> is different from <a>, so is <CTRL a>. The <CTRL> key is combined with other keys for special purposes.

In some books and manuals, the symbol * is used to represent the control key. For example, you may see ^a rather than <CTRL a>. As well, you may see a capital letter used (^ A). Regardless of which you see, it still means <CTRL a>. (You don't need to use a capital letter.) In this book, I will use only the straightforward <CTRL a> style.

At the bottom of the typewriter-key area is the long bar that corresponds to the spacebar on the typewriter. I will call this key <SPACE>.

Whenever you press a key, the keyboard sends a signal to the computer. When you press <SPACE>, the keyboard sends the signal for a character called space. It is important to understand that there are no empty gaps when you type. If you type:

    aa bb cc

you have typed eight characters; the third and sixth are space characters.

The concept of pressing keys that generate characters that you can't see is also important with the <ENTER> and <TAB> keys.

The <ENTER> key is on the right-hand side of the typewriter-key area. It is in the same place as the carriage return key on a typewriter. Indeed, it is sometimes referred to as such, but don't think of it that way. When you press <ENTER>, you are really sending a signal for a character called newline. Thus, if you type:

    aa bb cc
and then press

\texttt{<ENTER>}

you have typed nine characters. The third and the sixth are \texttt{spaces}, and the ninth is a \texttt{newline}.

To use a Xenix command, you type the command and then press \texttt{<ENTER>}. As you type the command, Xenix reads it, character by character, until it reads a newline. Then, it carries out the command. \textbf{We use the term ENTER to refer to typing a command and then pressing <ENTER>.} For example, if I ask you to enter:

\texttt{date}

I want you to type \texttt{date} and then press the \texttt{<ENTER>} key.

Near the top left-hand corner of the typewriter-key area, there is a key with two horizontal arrows on it. This is the \texttt{<TAB>} key. (Notice that it does not say "tab.")

When you press \texttt{<TAB>}, it sends the signal for a character called \texttt{tab}. This is a legitimate character, just like \texttt{space} and \texttt{newline}. When you are entering commands, Xenix automatically replaces a \texttt{tab} with six \texttt{spaces}. However, when you are creating a document, a \texttt{tab} is retained as a unique character that becomes part of the document.

In the top right-hand corner of the typewriter-key area, there is a key with a left-arrow. This is the \texttt{<BACKSPACE>} key. You use it to correct typing errors. Each time you press \texttt{<BACKSPACE>}, Xenix erases one character.

For example, if you misspelled the \texttt{date} command, you might enter:

\texttt{daxx<BACKSPACE><BACKSPACE>te}

This would have the same effect as entering:

\texttt{date}

Before we leave the typewriter-key area, I would like to point out a few keys that may be new to you.

First, Xenix uses three different quotation symbols, each with a different meaning (to be explained later). Just to the left of the \texttt{<ENTER>} key is the key that has the single and double quotation marks. In the top left-hand corner of the typewriter-key area is a key that contains a backwards quotation mark. I will refer to these symbols as double-quote, single-quote, and back-quote, respectively.

Also to the left of the \texttt{<ENTER>} key, just above the single-quote and double-quote, are the square brackets and brace brackets. On the same key as the back-quote is a curvy symbol (\texttt{~}) that you may not
have seen before. It is called a tilde. In the top row above the "6" is the caret (^), and just to the left of the <BACKSPACE> key is the vertical bar (|).

Last, there are two different slash symbols. Near the bottom right-hand corner of the typewriter-key area (below the "?"), is a slash that points upward to the right (/). This is called the slash. Near the top right-hand corner, below the vertical bar, is a slash that points downward to the right (\). This is called the backslash.

All of these special characters, including the more familiar ones (\$, &*, and so on), have meanings that you will learn later. The <ALT> key, in the bottom left-hand corner of the typewriter-key area, does not have a particular use with Xenix.

The Numeric Keypad

The numeric keypad is designed to allow you to enter numbers easily. There are keys for the ten digits <0>, <1>, <2>, <3>, <4>, <5>, <6>, <7>, <8>, <9>, for the decimal point <.>, the plus sign <+>, the minus sign <->, and the multiplication sign <*>.

The latter four symbols also appear in the typewriter-key area. They are placed in both areas for convenience, and you can use whichever you want. The symbol . is used for both a period and a decimal point, and - is used for both a hyphen and a minus sign.

Notice that the ten digits and the decimal point on the numeric keyboard are on the upper half of their keys. There are two ways to type these characters. You can either hold down the <SHIFT> key, as with the special characters in the typewriter-key area, or you can use the <NumLock> key.

The <NumLock> key is in the top row of the numeric keypad. It acts like the <CapsLock> key, but for the numeric keypad only. Normally, you will use the keys in the top row of the typewriter-key area to enter numbers. However, if you have a lot of numbers to enter, press <NumLock>, and use the numeric keypad.

When you press <NumLock>, the corresponding light (in the top right-hand corner of the keyboard) will come on. To return the numeric keypad to normal, press <NumLock> a second time. The light will go off.

The numeric keypad has other keys. The key labeled "Esc," in the top left-hand corner, is called the <ESCAPE> key. It is not used much; in particular, you do not use it when you enter commands. In the cases
where the <ESCAPE> key is used, it allows you to "escape" from something that is happening.

The <DEL> key, in the bottom right-hand corner, has a special use that will be discussed later.

The four keys with arrows are the CURSOR-CONTROL KEYS. They are used to change the position of the cursor. I will call them <CURSOR-LEFT>, <CURSOR-RIGHT>, <CURSOR-UP>, and <CURSOR-DOWN>. Do not use the cursor-control keys when you enter commands. These keys are for special situations. For example, if you are creating a document, you can use the cursor-control keys to move the cursor to various places in the document as you make changes.

Take care not to confuse <CURSOR-LEFT> with <BACKSPACE>. All <CURSOR-LEFT> does is move the cursor; <BACKSPACE> is the one to use when you want to erase a mistake.

The other keys of the numeric keypad—<ScrollLock>, <SysReq>, <HOME>, <PgUp>, <PrtSc>, <END>, <PgDn>, and <INS>—are not used with Xenix.

INPUT FROM THE KEYBOARD

What Happens as You Type?

As you type, Xenix reads each character and displays it on the screen. This is called ECHOING. Xenix echoes so fast that it seems as if your keyboard is connected directly to the screen. When you press a key and the character appears almost instantly, it is difficult to believe that Xenix is acting as a middleman.

In certain cases, Xenix will not echo what you type. For example, it does not echo your password as you type it (to prevent somebody from reading it over your shoulder).

Each time you type a character, Xenix collects it in a special area of memory. A temporary holding area like this is called a BUFFER. This particular buffer is called the KEYBOARD BUFFER.

When you press <ENTER>, it sends a newline character to the keyboard buffer. When Xenix sees this character, it takes the contents of the buffer and handles it in an appropriate manner. For example, if you have entered a command, Xenix carries it out.

Most of the time, you can ignore all these details. You can type what you want, press <ENTER> at the end of the line, and let Xenix do the
work. However, there are two ways in which understanding this system comes in handy.

First, since Xenix not only echoes what you type but collects it in a buffer, you can type as fast as you want and nothing will be lost. Thus, you can type ahead, even if Xenix is busy doing something else.

If you type while Xenix is displaying characters on the screen, the characters you type may be intermixed with the characters being displayed. However, since Xenix is collecting what you type in the keyboard buffer, nothing is lost. The only time that you can't type ahead is when you are entering your userid and password at the beginning of a work session.

The second way in which the buffer system is helpful is that it allows you to correct mistakes before your press <ENTER>. As you type, the position on the display screen is marked by a blinking underscore character, called the CURSOR. Xenix collects each character and moves the cursor forward. When you press <BACKSPACE>, Xenix deletes one character from its collection. Xenix then updates the screen display by moving the cursor one position to the left and erasing the character in that position.
CHAPTER 3

STARTING TO USE XENIX

INTRODUCTION
STARTING A WORK SESSION
THE COMMAND PROCESSORS
ENDING A WORK SESSION
INTRODUCTION

Every work session starts with typing information to connect your terminal to Xenix and ends with a command to disconnect the terminal. While you are working, you interact with a special program that reads and interprets your commands. This chapter explains how to start and stop a work session and how to understand the program with which you will be working.

New Words in This Chapter

LOG IN
to start a work session with Xenix

LOG OUT
to end a work session with Xenix

INSTALL [Xenix]
to set up the Xenix system for use with your computer

SYSTEM MAINTENANCE
the tasks that the system manager carries out to take care of the system

COMMAND PROCESSOR
a program that reads and carries out commands that are entered from the keyboard

SHELL
the Xenix name for a command processor

BOURNE SHELL
a shell developed by S. R. Bourne at Bell Labs; the most commonly used shell

C-SHELL
a shell often used by programmers and experienced users; the name comes from the fact that this shell has features in common with the programming language C; pronounced: "see-shell"

VISUAL SHELL
a shell that was designed to provide a visually pleasing environment; the visual shell gives inexperienced and casual users an easy-to-use interface to Xenix
Starting to Use Xenix

WINDOW
a box drawn on the screen in which information is displayed

MENU
a list of choices displayed on the screen

PROMPT
one or more characters displayed by a program to let you know that it is waiting for input

erase
the character produced by pressing the <BACKSPACE> key; erase instructs the shell to erase the last character you typed

kill
the character produced by pressing <CTRL-u>; kill instructs the shell to erase the entire line that you are typing

intr
"interrupt"; the character produced by pressing the <DEL> key; <DEL> is often used to stop a command or program before it is finished

eof
"end of file"; the character produced by pressing <CTRL-d>; eof signals to a program that is reading data that there is no more data; <CTRL-d> is often used to stop programs that read data from the keyboard; pronounced: "ee-oh-eff"

stop
the character produced by pressing <CTRL-s>; stop instructs the shell to pause the screen display

start
the character produced by pressing <CTRL-q>; start instructs the shell to continue with the screen display after you have sent the stop character

Before You Start

There are two ways in which you can access Xenix. First, you can use the keyboard and display of the main computer as your terminal. This will be the case if you are using Xenix on your own computer. (As I mentioned in Chapter 1, the keyboard and display of the main computer form the console.) The other way is to use a remote terminal connected
with the main computer, either by direct line or by phone. The main difference is that if you are using the console, you will have to turn on the computer and start Xenix.

In either case, you must first get your terminal connected with Xenix. Then you tell Xenix that you are ready to start. This latter process is called LOGGING IN. When you are finished working, you must tell Xenix that you are through. This is called LOGGING OUT.

STARTING A WORK SESSION

Starting from the Console

Before you can use Xenix, you must set it up properly on your computer. This is called INSTALLING Xenix. When you install Xenix, you set up the fixed disk, and then you copy the Xenix system from the master diskettes to the fixed disk. If you have not already installed Xenix, do so now. The instructions are in the IBM Installation Guide that comes with the system. Once Xenix is installed on your computer, you are ready to start.

Turn on the computer. If Xenix is properly installed, you should see the following message after a few moments:

```
xenix 286 boot
enter: device program
press enter for default: hd /xenix
```

Press <ENTER> to start Xenix. You will see some copyright information, followed by a message showing you how much memory was allotted to the different parts of the system. You will then see the message:

```
type ctrl-d to proceed with normal startup, (or give root password for system maintenance):
```

System maintenance is explained below. Go ahead with the normal startup. Press <CTRL-d> and Xenix will display the time and date followed by:

```
login:
```

You are now ready to log in.
Starting When Xenix Was Not Shut Down Properly

If your system was not shut down properly the last time it was used, Xenix will start somewhat differently. (The proper way to shut down Xenix is described below.) After you have pressed <ENTER> to start Xenix, you will see the message:

the system was not shut down properly, and the root file system should be cleaned (y/n)?

The “root file system” holds most of the Xenix system files. Xenix is asking for permission to examine the system and fix it if necessary. Enter y for “yes.” Xenix will proceed to “clean” the system in a six-phase process. It will display messages as it goes. You can ignore the messages, no matter how ominous they seem. When Xenix is finished, it will display:

***** file system was modified *****
** normal system shutdown **
press enter to reboot.

Don’t let this message scare you; everything is fine. Start Xenix by pressing <ENTER>.

Xenix will start as usual. The only difference will be that after you press <CTRL-d>, Xenix will automatically “clean” the part of the file system that serves the users. This is a five-phase process. Again, you will see messages that you can safely ignore. Once Xenix is through, you will see the usual log in message, and you can proceed.

System Maintenance

SYSTEM MAINTENANCE refers to the tasks that the system manager carries out; for example, registering new users. You need superuser privileges to carry out system maintenance. You obtain these privileges by logging in with the root userid and giving the superuser password. This is the password that you set when you install Xenix.

Everytime you start Xenix, you come to the point where you can either press <CTRL-d> to proceed with the normal startup or enter the root (superuser) password for system maintenance. The opportunity to enter the root password at this point is really just for advanced programmers doing special things. In all cases, press <CTRL-d> and start in the normal manner. If you do have system maintenance to perform, log in as superuser under the root userid.
Starting a Work Session 23

Starting from a Remote Terminal

You can either use Xenix from the console or from a remote terminal. Your terminal may be a real terminal, or it may be a computer emulating (pretending to be) a terminal. For example, you may use an IBM Personal Computer to emulate an IBM 3101 Terminal.

If you are using a real terminal, turn it on. If there is a switch to make the terminal “on-line” or “off-line,” make sure that it is set for “on-line.” If you are using a computer, turn it on and start the emulation program.

If your terminal is wired directly to the main computer, then turning on the terminal should establish communication automatically. If you are using a phone line, you will have to dial the appropriate number. In either case, as soon as you are connected to Xenix you will see:

login:

You are now ready to log in.

Logging In

You are ready to log in when you see:

login:

Enter your userid. If you want to log in as the superuser, enter root. After you enter your userid, you will see:

password:

Enter your password. Notice that it is not echoed as you type. This prevents anyone from finding out your password by looking over your shoulder as you log in.

If you enter both your userid and password correctly, Xenix will log you in to the system. If either your userid or password were incorrect, Xenix will make you repeat the whole process. Xenix does not check the userid until after you have entered the password. This means that Xenix will ask for a password even if the userid is incorrect.

Xenix has a built-in mail system that is used to send messages. As part of the log in procedure, Xenix will tell you if you have any messages waiting. When a new user is registered, Xenix automatically sends mail welcoming him or her to the system. Thus, the first time you log in, you will have mail waiting for you. The mail system is explained in a later chapter; for now, you can ignore it.
If You Are Having Trouble with Your Terminal

If you have trouble logging in with a terminal that someone has been using, it may be that they have forgotten to log out. If you suspect that this is the case, enter the following command:

`who am i`

If someone is logged in at the terminal, Xenix will display that person's userid. In this case, you will have to log that person out before you can log in. (Logging out is explained below.) Of course, before you take over the terminal, make sure that it has really been abandoned—the user may have just walked away for a few minutes.

If your terminal won't work properly and this is not the problem, press <DEL> a few times. If this doesn't work, turn the terminal off and on. (Do not do this with the console—only with a regular terminal.) Next, check to make sure that the terminal is properly connected and that the Xenix system is running.

If your terminal is properly connected and Xenix is running, and you still can't start properly, you will have to ask someone for help.

THE COMMAND PROCESSORS

The Shells—Command Processors

A program that reads commands from the terminal and carries them out is called a COMMAND PROCESSOR. The Xenix name for a command processor is a SHELL. The shell is the part of Xenix that reads what you enter from the terminal and interprets it appropriately.

Since the shell processes your commands, it provides the environment in which you work. Xenix comes with three shells, each with different characteristics. You can choose how you will work with Xenix by picking the shell that suits your needs.

When you are given a userid, the system manager specifies what shell you are going to use. When you log in, Xenix automatically starts this shell. You can enter a command to change to one of the other shells any time you want, but usually there is no need to do this. If you decide that you want to use a different shell permanently, have the system manager make the appropriate change for you.

The three shells are the BOURNE SHELL, the C-SHELL, and the VISUAL SHELL.
The Bourne shell is named after its designer, S. R. Bourne of Bell Labs. The Bourne shell is the original shell and is the one most people use. This is the shell that you would use unless you had a reason to use one of the others.

The C-Shell (pronounced: “see-shell”) is a newer shell, often used by programmers and other experienced users. The rules for using this shell are similar to those that are used with the programming language C (the main computer language used with Xenix), hence the name C-Shell.

The visual shell gets its name from the fact that it uses the entire display screen to produce a visually pleasing interface to Xenix.

The Differences between the Shells

The three shells provide different ways of interacting with the same Xenix system. However, there is a fundamental difference between the visual shell and the other two shells.

The visual shell provides a visually pleasing environment. At all times, you see a screenful of information. This includes a large box, called a WINDOW, in which information is displayed, and a number of command names near the bottom of the screen.

The command names at the bottom of the screen offer a choice. Such a list of choices is called a MENU. With the visual shell, you enter a command by selecting it from the menu.

As you work, the menu changes to offer choices appropriate to what you are doing. Similarly, the information in the window changes as you select various commands. For example, when you start, the menu offers a choice of the most common commands, and the window displays the names of your files.

The visual shell also comes with an extensive help system that you can draw on as you work.

The Bourne shell and the C-Shell are completely different from the visual shell. Rather than automatically displaying information and offering menus, these shells simply wait for you to enter a command. After you press <ENTER>, the shell carries out the command and then waits for another.

Programs that wait for you to enter information usually display one or more characters, called a PROMPT, to let you know that they are waiting. When you log in and you see the prompt, you know that the log in procedure is finished and that the shell is waiting for you to enter a command.
The Bourne shell displays a dollar sign ($) and the C-Shell displays a percent sign (%). When you log in as the superuser, the prompt is the number sign (#) for either shell. This reminds you that you are the superuser.

When you work with the visual shell, you constantly see a screenful of information. With the other shells, you are prompted only with a $ or a %. Since there are no menus, it is up to you to remember the various commands and to enter the one you want. Thus, with the Bourne shell and C-Shell, you are on your own. Unfortunately, these shells do not have a help facility, so you will have to depend on this book and the IBM manuals for reference.

The Bourne shell and C-Shell have two main advantages over the visual shell. First, they are more flexible and powerful. Because you do not have to select from a menu, you can enter any Xenix command, with any options, at any time. As well, these shells have important features that are missing from the visual shell.

Second, once you develop a repertoire of commands, it is much faster to specify what you want rather than to select choices from a menu. Thus, experienced users prefer the Bourne shell or C-Shell.

And what are the differences between the Bourne and the C-Shell? The Bourne shell is the most widely available shell. If you know how to use the Bourne shell, you will be able to use any system in the Unix family.

The C-Shell is not as common. For example, when you buy Xenix from IBM, you automatically get the Bourne shell and the visual shell. You do not get the C-Shell unless you buy the optional Software Development System.

The C-Shell has certain features that are not present in the Bourne shell. If you use Xenix a lot, these features provide a preferable working environment. However, the C-Shell and the Bourne shell are similar enough that once you learn one, it is easy to learn the other.

Choosing Your Shell

Use the visual shell if you plan on being a casual user or if you will be using Xenix for one specific purpose (say, to use a spreadsheet program). Use the C-Shell if it is available on your system and you plan to write programs or to do a fair amount of work. Otherwise, use the Bourne shell.

In this book, I will assume that you are using the Bourne shell. If you want to use the visual shell, read through the book to learn how
Xenix works, but don't bother with the examples. If you want to use the C-Shell, use the book in a normal fashion. I will point out the places where the Bourne shell differs from your shell.

**Special Signals from the Keyboard**

(Note: This section does not apply to the visual shell, which uses the keyboard in its own way.)

The Bourne shell and the C-Shell recognize six special signals that you can use to control what is happening. Each signal has a key that you press to send that signal.

The first signal is **erase**. Its key is the `<BACKSPACE>` key. The **erase** signal causes the shell to erase the last character that you typed. Thus, you use `<BACKSPACE>` to correct mistakes.

The next signal is **kill**. Its key is `<CTRL-u>`. **kill** erases the entire line that you are typing. If you make so many mistakes on a line that it is too much trouble to correct them with `<BACKSPACE>`, press `<CTRL-u>`, and retype the entire line.

The third signal is **intr** ("interrupt"). Its key is `<DEL>`. On some terminals without a `<DEL>` key, use the key called `<RUBOUT>`. This signal interrupts what is happening. Use `<DEL>` to stop a command before it finishes.

For example, say you have just entered a long, complicated command. As soon as it starts, you realize that it is not doing what you wanted. Press `<DEL>` to stop it.

The fourth signal is **eof** ("end of file"). Its key is `<CTRL-d>`. When a program is reading information from the keyboard, you need a way to signal the end of the information. This is what **eof** does.

For example, say you use the **sort** command to read information from the keyboard, sort it, and display the results. You enter the information line by line. After you enter the last line, you press `<CTRL-d>` to indicate the end.

The **eof** signal is sometimes used to stop a program. For example, the **bc** command starts a program that performs calculations. This program waits for you to enter an arithmetic expression from the keyboard. The program displays the answer and then waits for more input. You can enter as many expressions as you want, one at a time. When you are finished, you press `<CTRL-d>` to signal the program that there is no more input, and the program stops.

The last two signals are called **stop** and **start**, and they work together. The keys are `<CTRL-s>` for **stop** and `<CTRL-q>` for **start**.
They are used to control the displaying of information. <CTRL-s> pauses the screen display, while <CTRL-q> restarts it.

If you use a program that displays a lot of information quickly, most of it will go by so fast that you won’t be able to read it. You can press <CTRL-s> any time you want to stop the display. When you are finished reading what is on the screen, press <CTRL-q> to continue.

ENDING A WORK SESSION

Logging Out

When you are finished working, you tell Xenix by logging out. It is important that you log out, rather than simply turn off your terminal or computer. Xenix has certain housekeeping chores that it must perform every time a user leaves the system.

Logging out is easy. Remember that while you are working, the shell is waiting to read your next command from the keyboard. All you have to do is press <CTRL-d>. This sends the eof signal to the shell, telling it that there is no further input. The shell then stops itself, at which time Xenix terminates your work session.

If you are in the middle of using a program, you will first have to signal that you are finished. In this case, you may have to press <CTRL-d> more than once to log out.

If you are using the visual shell, you log out by selecting the quit command. The shell will ask you to confirm that you really want to log out. Enter y (for “yes”).

If you are using the C-Shell, it may be set up so that you can’t log out by pressing <CTRL-d>. This is to prevent you from accidentally logging out when you are stopping another program. In this case, you can log out by using the logout command that is only for the C-Shell. Simply enter:

    logout

You are now logged out.

Stopping Xenix

Xenix is designed to run continuously. When the last user has logged out, Xenix waits for someone else to log in. However, if you are
using Xenix on your own computer, you will certainly want to turn it off when you are finished. You may also want to stop Xenix in order to start another operating system.

It is important that you do not just turn off the power. Before Xenix can stop, it needs to perform certain chores. If you turn off the power abruptly, Xenix cannot perform these chores, and it may affect the integrity of the file system.

When you want to stop Xenix, follow these steps:

1. If you are logged in as a regular user, log out.
2. Log in as the superuser (userid root).
3. To stop Xenix, use the shutdown command; enter: `shutdown`.

Xenix will ask you how many minutes to wait before shutting down the system. If you are the only user, specify 0 (zero), so Xenix will shut down right away. If there are other users, specify at least five minutes to give them a chance to finish what they are doing and to save their work. Xenix will send each user a message to warn of the impending shutdown.

When the waiting time has elapsed, Xenix will shut down the system in an orderly and cautious manner. When the process is complete, you will see the message:

```
** normal system shutdown **
press enter to reboot:
```

You can now turn off the computer.

If some problem prevents you from using the `shutdown` command, you can halt the system immediately by using the `haltsys` command.

Enter: `haltsys`

This command stops everything—including all work in progress. If there are any other users, they will be logged out right away. Therefore, do not use this command unless the `shutdown` command does not work properly.
CHAPTER 4
LEARNING TO USE XENIX

INTRODUCTION
WHERE TO FIND REFERENCE MATERIAL
WHAT COMMANDS LOOK LIKE
HOW COMMANDS ARE DESCRIBED
INTRODUCTION

Xenix commands have a particular structure that you need to understand. This chapter teaches you about that structure and about how it is described in manuals and books. In order that you may learn as easily as possible, the chapter first discusses the various places that you will find Xenix reference material, including the reference manuals that come with the system.

New Words in This Chapter

OPTION
the part of a command that gives you control over how the command carries out its task

ARGUMENT
the part of the command that specifies information that the command needs to carry out its task

SYNTAX
a description of how a command must be entered.

WHERE TO FIND REFERENCE MATERIAL

The Xenix Reference Manuals

In this book, I will introduce you to the most important commands and teach you how to use them. However, your reading should not end with this book. There are a number of manuals that come with Xenix, and you should make sure that they are handy while you are working.

The Xenix system is sold as three separate products: the Xenix operating system and two optional systems, one for text formatting (preparing documents) and one for software development (writing programs). In this section, I will describe the manuals that come with the operating system.

The most valuable manual is the Xenix Command Reference. This manual has three sections. The first and largest lists almost all the Xenix commands. If you have a problem with any command, you can
look it up. Consider this manual to be the definitive reference. Keep it
next to your computer or terminal at all times.

The second section of the Xenix Command Reference explains
system maintenance commands and information. The third section
explains the formats of the system files. You will probably have little
need for these sections.

The other manuals are:

- Xenix Basic Operations Guide
- Xenix Visual Shell
- Xenix Installation Guide
- Xenix System Administration

The Basic Operations Guide is a general reference. Use it to
supplement this book and the Command Reference. The first few
sections are the IBM version of an introduction to Xenix.

The Visual Shell manual is useful only if you are going to use the
visual shell. Otherwise, this is a book you can safely ignore.

The Installation Guide and System Administration manuals are for
system managers. If you are not the system manager, ignore these
books. If you are using Xenix on your own personal computer then you
are, of course, the system manager and you will need these manuals.

Use the Installation Guide to help you set up Xenix. With the
System Administration manual, read the first few chapters to comple­
tment this book. Then skim the rest of the manual to get an idea of what
it offers, and put it away to refer to when you have a problem in this
area.

**More Ways to Learn Xenix**

As I mentioned earlier, Xenix is a member of the Unix family of
operating systems. Traditionally, Unix documentation has been poor
with respect to new users. Most people learned Unix from someone else
and used the manuals only for reference.

Although you have this book and the IBM manuals, you will find it
helpful to learn from other people. Try to find an expert or two who will
answer your questions as you are learning. You might also want to look
in your area for a Xenix User's Group.

Make sure that you set some time aside to experiment with Xenix.
Look through the Xenix Command Reference and try out various
commands. There are so many commands and variations that you can
learn a lot by purposeful exploration. In the long run, expanding your horizons will save you time, so don't feel bad about neglecting your work occasionally to go exploring.

Because Xenix is so large and difficult to master, it is important to develop good habits. First, use the reference material. When you are not sure about something, take a moment to look it up, either in this book or in the Command Reference manual.

Second, go out of your way to learn new things. It is possible to use only a small number of commands and get by, but you are better off expanding your skills by adding new commands to your repertoire.

Here is an example: You can use the editor program vi to create and modify files of information. When you learn vi, you will see that it has too many commands to learn all at once. Some people learn a few basic commands, enough to get by, and stop there.

When you use vi, don't make this mistake. Every now and then, stop your work and teach yourself a few new commands. As you expand your skills, you will find your work becoming easier and more pleasant.

One last piece of advice: Xenix is very similar to Unix. If you find a Unix expert or a Unix reference manual, most of what you hear and read will hold for Xenix. If you become a serious Xenix user, you may want to buy the Unix Programmer's Manual. This is a two-volume set put out by Bell Labs and published by Holt, Rinehart and Winston.

WHAT COMMANDS LOOK LIKE

The Form of Xenix Commands

Xenix commands have three parts:

• the name of the command
• options
• arguments

A command always has a name, but it may or may not have options and arguments.

You can enter a command any time you see the shell prompt ($ for the Bourne shell, % for the C-Shell, # for superuser). To enter a command, type the name of the command, followed by any options, followed by any arguments (in that order), and then press <ENTER>.
Here is an example of a command:

```bash
lc -l book
```

In this example, `lc` is the name of the command, `-l` is an option, and `book` is an argument.

When you enter a command, you must separate each part of it by one or more `space` or `tab` characters. For example, the following three commands are equivalent:

```bash
lc -l book
lc -l book
lc <TAB> -l <TAB> book
```

However, you cannot use spaces or tabs in the middle of an option or argument. So the following commands will not work properly:

```bash
lc book
lc -l book
lc -l bo ok
```

Each command performs a particular task. OPTIONS give you control over how the command carries out the task, and ARGUMENTS specify information that the command needs to do its work. (The term `argument` is borrowed from mathematics. However, the original meaning is highly technical. Better just memorize the term as a new word.)

You can enter more than one command on the same line by separating them with semicolons. Xenix will execute the commands, one by one. Here is an example:

```bash
date; lc -l book; more file1
date ;lc -l book ;more file1
date ; lc -l book ; more file1
```

Think of the semicolons as command separators.

## Command Options

Many commands have options that allow you to control the command. For example, the `lc` command displays information about files. The options for this command let you specify what information is to be displayed, and in what form. Consider this example:

```bash
lc -l book
```

In this command, the option `-l` specifies that the information is to be
displayed in the “long” format.

Options always start with the “-” character. When we talk about commands, we pronounce “-” as “minus” (even though it is clearly not being used as a minus sign). For example, if you wanted to tell someone to use the command `lc -l book`, you would tell them to use the “minus-ell” option.

As the name implies, options are optional. You will often enter commands without any options. For example:

```
lc book
```

On the other hand, many commands have more than one possible option. (The `lc` command has 22.) You can enter as many options as make sense for the command you are using. If you enter more than one option, you can put each one after a separate “-”, or you can put more than one after the same “-”.

For example, the following commands are equivalent:

```
lc -FIR book
lc -F -l -R book
lc -F -IR book
```

Most options use one lower-case letter. However, some use upper-case letters and some use numbers. Make sure that you do not mix them up. For instance, with the `lc` command, `-f` and `-F` are two completely different options; as are `-l` and `-1`.

When you use a command with more than one option, you can specify the options in any order you want. Just make sure that all the options come after the command name and before the arguments. The following commands are all equivalent:

```
lc -F1RI book
lc -R1F1 book
lc -l -l -F -R book
lc -RF -l -l book
```

Sometimes the options letters make sense—for example, `-l` for “long format”—but often they do not. As well, there is usually no consistency between commands. For example, `-l` means one thing with the `ls` command, and something completely different with the `wc` (word count) command.

Because of this, do not try to memorize all the options for every new command that you learn. Rather, concentrate on one or two important options, and look up the others when you need them.
Command Arguments

The purpose of arguments is to specify information that varies each time you use the command.
For example, the \texttt{diff} command compares two files of information. You specify the names of the file each time you enter the command. This allows you to use the command with any files you want.

Here are some examples:

\begin{itemize}
  \item \texttt{diff oldrecords newrecords}
  \item \texttt{diff file1 file2}
  \item \texttt{diff accounts supplies}
\end{itemize}

Each of these commands has two arguments. However, the value of the arguments is different in each case. It is as if you are using the arguments to pass information to the command; in this case, the names of two files.

When you learn a new command, you need to learn what arguments the command uses and what sort of information those arguments should contain. Sometimes arguments are optional, and sometimes they are required.

HOW COMMANDS ARE DESCRIBED

Command Descriptions and Syntax

A description of how a command must be entered is called the \textsc{Syntax} of that command. It is customary to use certain conventions when we write down the syntax of a command. In this book, I will make sure that I use the same conventions that you will find in the IBM manuals. They are as follows:

\begin{itemize}
  \item square brackets \texttt{[ ]} enclose optional parts of the command
  \item anything not in square brackets is obligatory
  \item italics indicate arguments; everything in italics must be given a value when you enter the command
  \item \ldots indicates arguments that can be repeated
\end{itemize}

Here are some examples of command descriptions:
**pwd**

The name of the command (pwd) is obligatory. There are no options or arguments.

**cp file1 file2**

The name of the command (cp) is obligatory. There are no options. There are two obligatory arguments. When you enter the command, you must enter a value for each of the arguments. Here are two examples of this command:

```
cp newfile oldfile
```

In the first command, file1 has the value `newfile`, and file2 has the value `oldfile`. In the second command, file1 has the value `original`, and file2 has the value `duplicate`.

**cat [-us] file...**

The name of the command (cat) is obligatory. There are two options. There is one obligatory argument that can be repeated. Here are some examples of this command:

```
cat -u -s file1
```

In the first command, two options are specified. The argument file has the value `file1`. The second command is equivalent to the first. (Remember, options can be specified separately or together, in any order, as long as they come before the arguments.) The third command has one option. The argument file is repeated three times and has the values `file1`, `file2`, and `file3`. The last command has no options. The argument file has the value `file1`.

**ls [-logtasdruf] name...**

The command name (ls) is obligatory. There are 11 possible options. There is one argument that may be repeated. This command has the same structure as the last example. The only difference is that there are more options.

**news [-ans] [items]**

The command name (news) is obligatory. There are three options. There is one optional argument. Here are some examples of this command:
The Commands in This Book

Xenix has more commands than you need to learn. As well, many commands offer more options than you will probably use.

In this book, I have selected the commands and options that are most useful. When I introduce a command, I will leave out the less important options. For a complete exposition of every command and every option, the Xenix Command Reference manual remains the definitive authority.
CHAPTER 5

COMMANDS THAT YOU CAN USE RIGHT AWAY

INTRODUCTION
FINDING OUT INFORMATION ABOUT THE SYSTEM
COMMUNICATING WITH OTHER USERS
FINDING OUT INFORMATION ABOUT TIME
COMMANDS THAT USE STRINGS
CUSTOMIZING YOUR WORKING ENVIRONMENT
TWO MATHEMATICAL COMMANDS
INTRODUCTION

Some Xenix commands are easy to learn, and you can use them as soon as you log in for your first time. This chapter covers a number of these commands. Follow along with the examples on your own terminal as you read. You will find these commands not only easy to use but interesting and useful. This chapter also explains two important topics: strings and single-quotes.

New Words in This Chapter

ECHO
to write information exactly as it was read

STRING
a sequence of characters

FACTOR
a whole number that evenly divides into another whole number; for example, 12 has six factors: 1, 2, 3, 4, 6, and 12.

PRIME NUMBER
a number that has no factors except 1 and itself; for example, 13 is a prime number while 12 is not

PRIME FACTOR
a factor that is a prime number; for example, of all the factors of 12 (1, 2, 3, 4, 6, and 12), only 2 and 3 are prime factors

FINDING OUT INFORMATION ABOUT THE SYSTEM

The news Command

The news command displays the system news. The system news is updated by the system manager. If you are the system manager, there won't be any news unless you set it up.

The syntax of the news command is:

```
news [-ans] [item...]
```
If you enter the command with no options, Xenix will display all the news items that have been added since you last displayed the news. For example:

```
news
```

If you specify the `-n` option, `news` will display the name of each new item in the file that you have not yet seen. (`n` stands for “new.”) For example, if you enter:

```
news -n
```

you may see something like this:

```
news: picnic passwords
```

In this case there are two news items you have not seen: one about a picnic and one about passwords.

Once you know the names of the news items, you can display the ones you want by entering their names as arguments. For example, to display the item named `picnic`,

```
news picnic
```

The `-s` option will display the number of news items that you have not yet seen. For example, if you enter:

```
news -s
```

you might see:

```
2 news items
```

If you want to see all the news items, including all those you have seen before, use the `-a` option. (`a` stands for “all.”)

---

**The logname Command**

The `logname` command displays the userid that is logged in to the terminal you are using. The syntax is:

```
logname
```

Usually, all this command will do is tell you your own userid. However, if you come upon a terminal that looks abandoned, you can use this command to see who is logged in. This command is also useful if you have more than one userid and you forget which one you used to log in.
42 Commands That You Can Use Right Away

The **uname** Command

This command displays information about your Xenix system. The syntax is:

```
uname [-r]
```

There are other options, but they are not really necessary. If you enter the command with the `-r` option, you will see something like:

```
3.0
```

This is the release number of the operating system. In this example, the message means you are using Xenix release 3.0. A higher release number means that you are using a newer version of Xenix.

COMMUNICATING WITH OTHER USERS

The **who** Command

This command displays all the userids that are logged in. Beside each userid is the name of the terminal being used. The syntax is:

```
who [am i] [are you]
```

Usually, you will use this command without an argument:

```
who
```

A typical result is:

```
  harley     console May 31 21:59
  dianne      tty01 May 31 22:02
```

In this example, there are two userids logged in. **harley** is using the console and logged in at 9:59P.M. on May 31. **dianne** is using **tty01** and logged in at 10:02P.M. on May 31. Notice that Xenix uses a 24-hour clock.

If you want to display this information about your terminal only, you can enter either:

```
  who am i
  who are you
```

These two forms of the `who` command are equivalent. Like the `logname` command, they are useful if you come upon an abandoned terminal.
The finger Command

If you want to find out more information about a userid, use the finger command. The syntax is:

```
finger [userid...]
```

This command displays information about each userid that you specify, even if they are not currently logged in. If you do not specify any arguments, the command will display information about all the users that are currently logged in.

Here is an example:

```
finger dianne
```

The following information is displayed:

```
Login name: dianne
In real life: Dianne MacLeod
Directory: /usr/dianne
Shell: /bin/vsh
No Plan.
```

This information shows that the userid dianne is registered to Dianne MacLeod. (Of course, anyone who knew her password might be logged in with the same userid.) The last few letters of the shell entry show that she uses the visual shell. The directory entry shows her home directory. (This is explained in Chapter 7.) The last line “No Plan.” is not important. (See the Command Reference manual for details.)

The write Command

This command allows you to use your terminal like a two-way radio to talk with another user. The syntax is:

```
write userid [tty]
```

To establish communication with someone, enter the command with the userid of the person to whom you want to talk. For example if you want to talk with the person whose userid is lisa,

```
write lisa
```

If the same userid is logged in to more than one terminal, use the who command to check which terminals are being used. When you enter the write command, you can specify a terminal name after the userid. For example, if lisa is logged in to both tty01 and tty02, you might enter:
write lisa tty02

The write command will signal the other user by beeping and displaying a message showing your userid and terminal. For example:

message from michael console...

At this point, the two terminals are connected. Every line entered on one terminal is automatically displayed on the other. When you are finished, either person can enter <CTRL-d> (end of file) to end the command.

Once you enter a write command, you will not see anything until the other person answers you. If it has been a long time, and you suspect that the person is away from the terminal, you can cancel the command by pressing <DEL>.

Whenever either person types a line, it is automatically displayed on the other terminal, as soon as the <ENTER> key is pressed. In order to keep the messages from getting mixed up, Xenix users generally follow a few rules to make sure that only one person at a time sends a message. Here is how it works:

Both people take turns entering messages—each message being as many lines as necessary. You signal the end of each message by entering -o-. This stands for “over” (as if you were sending messages by radio). This lets the other person know that you are now waiting to receive a message. When the conversation is over, the last person to send a message enters -oo-, for “over and out,” and presses <CTRL-d> to end the command.

Here is a sample conversation. michael, at the console, enters:

write lisa

The terminal where lisa is logged in beeps and displays:

message from michael console...

lisa enters:

hello michael, what do you want? -o-

michael enters:

I just wanted to know if you are going to Debbie's wedding? -o-

lisa enters:

I already did, it was last week. -o-
michael enters:

    oh..well, that's all for now. -oo- <CTRL-d>

Now that michael has pressed <CTRL-d>, the conversation and the command are terminated.

The mesg Command

Sometimes it will be inconvenient for you to receive messages. For example, you may be displaying a large amount of important information, and you don't want to be interrupted by messages suddenly appearing on your display screen. In this case, you can use the mesg command to deny permission to anyone to send you messages. The syntax is:

```
mesg [n] [y]
```

n stands for "no" and y stands for "yes." To keep anyone from sending you messages,

```
enter: mesg n
```

This is like taking your phone off the hook. If someone tries to use the write command to contact you, they will see:

```
write: permission denied
```

When you want to make your terminal available again,

```
enter: mesg y
```

If you want to check the current status, enter the command without an argument:

```
mesg
```

mesg will tell you if it is set to n or y.

FINDING OUT INFORMATION ABOUT TIME

The date Command

To display the date and time, use the date command. This command has options, but you will normally not need them. Ignoring the options,
the syntax is:

```bash
date
```

Here is an example of the output of the `date` command:

```
sun jun 1 23:46:13 pdt 1986
```

The date is Sunday, June 1, 1986, and the time is 11:46P.M. (and 13 seconds), Pacific Daylight Time.

The time is only as accurate as it was set. It is the job of the system manager to set the time correctly.

Use the `date` command when you want either the date or the time. There is no separate “time” command. (There is a command called `time`, but it is part of the software development system, and it has a completely different purpose.)

---

The `cal` Command

The `cal` command displays a calendar. The syntax is:

```bash
cal [month] [year]
```

If you enter this command with no arguments, it will display a calendar for the current month, plus the previous and next month. For example:

```bash
cal
```

If you specify a particular year, you will get the calendar for that year. For example:

```bash
cal 1952
```

You can specify any year from 1 to 9999, but you must enter the full year. For example, `cal 52` refers to the year 52 A.D., not 1952. `cal 1952` refers to 1952.

If you only want the calendar for a particular month, specify it before the year. For example:

```bash
cal december 1952
```

You can abbreviate the month by using enough letters to identify that month uniquely. For instance, `cal d 1952` will do for “december,” but `cal may 1952` is necessary for “may” to distinguish it from “march.” You can also refer to a month by number. For example, the following commands are equivalent:
If you specify a month but not a year, the `cal` command will use the current year. For example, to display the calendar for December of the current year,

```
enter: cal december
```

Do not use a number for the month in this case, or it will be interpreted as a year. For example:

```
cal 12
```

displays the calendar for the year 12 A.D., not for December of the current year.

*Note:* There is a command called `calendar` that is completely different. Do not confuse the two commands.

**COMMANDS THAT USE STRINGS**

The `echo` Command

This command is well named. All it does is display its arguments. The `echo` command has a few options that I will explain later in the book. For now, consider the syntax to be:

```
echo [argument...]
```

For example, if you enter:

```
echo hello, how are you?
```

you will see:

```
hello, how are you?
```

Sometimes the word ECHO is used as a verb, to indicate writing information exactly as it was read. For example, we might say that the `echo` command echoes its arguments to the display. Or we might say that a certain program echoes its input as it reads.

Of course, you can use the `echo` command to practice writing various messages to the display. However, there is a more important use that I will explain in Chapter 20.
48 Commands That You Can Use Right Away

The banner Command

Conceptually, the `banner` command is similar to the `echo` command in that it echoes its arguments. However, this command produces large letters. The syntax is:

```
banner argument...
```

Each argument can be up to ten characters long. Try this example:

```
banner here is a big message 123456789012345
```

Notice that only the first ten characters of the last argument were displayed.

Later in the book, you will learn how to send the output of a command anywhere you want. In Chapter 15 you will learn how to send the output of `banner` to the printer to make your own signs.

When other commands send information to the printer, they sometimes use the `banner` command to print an identifying name at the front of the printout.

Strings and Single Quotes

A STRING is a sequence of characters. For example, here are five strings:

```
abcde  ABCDE  12345 bg$&A();;;  hello?
```

When you enter a command, the shell considers it to be a series of strings, separated by either space or tab characters. The first string is the name of the command, and the other strings are options and arguments. For example, if you enter the following command, you have entered four strings:

```
banner hi to you
```

Most of the time, you can forget that a command is made up of strings, but occasionally you must pay them particular attention. This happens whenever you want to enter a string that contains a character that has a special meaning to the shell.

The simplest case occurs with a string that contains space characters. For example, if you enter the command in the last example, `banner` will display each argument separately. But what if you want to display the three words together? The solution is to use one long argument, consisting of the three words. To indicate this, enclose the words in single-quotes:
Commands That Use Strings

**banner 'hi to you'**

When a string is enclosed in single-quotes, it tells the shell to interpret all the characters literally, not in a special way. In this case, the **space** characters do not separate arguments; they are mere spaces.

The result is that you have passed **banner** one long argument, rather than three short ones. Enter both examples for yourself, to appreciate the difference.

Besides the **space** character, there are a number of other characters that have special meanings: the **tab**, **newline**, `<CTRL>` characters, and most of the special characters: *, ?, &., and so on. Whenever you want these characters to be interpreted literally as part of a string, you must put them within single-quotes. For example:

```
echo here is an ampersand '&'
```

displays:

```
here is an ampersand &
```

This shell is a complicated program. Altogether, it recognizes three different ways to quote strings, each with its own use. The other quoting methods are explained later in the book.

**Strings That Contain newline Characters**

You can enter a string that contains a **newline** character, by enclosing the string in single-quotes and pressing `<ENTER>` where you want the **newline**.

For example, you can have **echo** display more than one line at a time by putting **newline** characters where you want to start a new line:

```
 echo 'line one<ENTER>line two<ENTER>line three'
```

When **echo** displays the string, it will break the line whenever it encounters a **newline**. You will see:

```
line one
line two
line three
```

When you press `<ENTER>` in the middle of a string, the cursor will move to the beginning of the next line on the display, as it usually does. However, the shell will not display the usual prompt. Instead, you will see an alternate prompt: `>` (the greater-than sign). This lets you know that the shell is still collecting characters as part of one long command.
So, in this case, as you enter the command, your screen will look like this:

```
$ echo 'line one
   line two
   line three'
```

If you are using the C-Shell, it works the same way, except you must type `<SPACE>\` (space backslash) before any `newline` that appears inside of a string. So you would enter:

```
echo 'line one \<ENTER>line two \<ENTER>line three'
```

When you press `<ENTER>` within a string, the C-Shell, unlike the Bourne shell, does not display an alternate prompt. So your display will look like this:

```
% echo 'line one
   line two
   line three'
```

The result will be the same.

**CUSTOMIZING YOUR WORKING ENVIRONMENT**

The `setkey` Command

If you are using an IBM PC AT computer, you can set the function keys by using the `setkey` command. The syntax is:

```
setkey keynumber string
```

The first argument is the number of a function key, from 1 to 10. The second argument can be any string. For example:

```
setkey 1 date
```

The `setkey` command assigns the string to the specified function key. Thereafter, when you press that key, the string is displayed and collected in the keyboard buffer just as if you had typed it from the keyboard.

For instance, say you have entered the above `setkey` command. Every time you press `<F1>`, the string `date` will be displayed:

```
date
```

All you have to do is press `<ENTER>` to enter the command.

As you remember, you can enter more than one command on a line if you separate the commands with semicolons. Here is an example of setting `<F2>` in this way:
Customizing Your Working Environment

You can now enter both commands by pressing <F2> <ENTER>. Notice that because the string contains space characters and a semicolon (which has a special meaning), it must be enclosed in single-quotes.

Of course, it would be more convenient if you didn't have to press <ENTER>, each time, to enter the sting. All you have to do is add a newline character to the end of the string:

```
setkey 2 'logname ; date'
```

Now, you can enter the two commands just by pressing <F3>. As you enter the setkey command, the shell will display the alternate prompt, and you will see:

```
$ setkey 3 'logname ; date
>
```

With the C-Shell, remember to type <SPACE>
before the newline:

```
setkey 3 'logname ; date \<ENTER>'
```

The setkey command can be used for any string, not just a command. For example, say you have a friend with a userid that is difficult to remember, like mxyzptlk. You can assign the string to a function key:

```
setkey 4 mxyzptlk
```

Whenever you want to type the userid, use the function key. For example:

```
write <F4>
```

If you want to change the string assigned to a function key, simply enter a new setkey command. What if you just want to remove the assignment without specifying a new one? From the syntax of the command, you can see that you always have to enter both arguments. So, enter an empty string for the second argument. For example:

```
setkey 4 ''
```

(Note: This example uses two single-quotes, not one double-quote.)

One last thing: Function key assignments only last until you log out.

The passwd Command

The passwd command allows you to change your password. The
syntax is:
52 Commands That You Can Use Right Away

**passwd userid**

Normally, you enter this command without an argument:

`passwd`

Xenix will ask for your current password. If you enter it incorrectly, you will be denied permission to change it. (This prevents someone from changing your password if you walk away from your terminal for a minute.)

Once you enter the correct password, Xenix will ask you to enter the new one. You can use any characters you want. It is a good idea to use some combination that cannot be guessed easily, such as `8%%%yy1`. You should specify at least five characters. If you enter more than eight characters, Xenix will use only the first eight.

The `passwd` command will only let you change the password for the userid you used to log in. (The one displayed by the `logname` command.) However, if you are the superuser, you can change the password for any userid, including the one for the superuser itself (userid `root`). In this case, specify the userid as an argument. For example, to change the password for userid `cimarron`:

`passwd cimarron`

If you are working on a small system, you may find it convenient to dispense with passwords. If so, log in as superuser and change each password. When it comes time to specify the new one, simply press `<ENTER>`. Xenix will change the password to an "empty" password. This means that when you log in with that userid, Xenix will not ask for a password. In the interests of system security, only the superuser can specify empty passwords.

It is a poor idea to dispense with the superuser password. Even if you are the only one using the system, you might accidentally log in as superuser and cause damage.

**TWO MATHEMATICAL COMMANDS**

**The random Command**

This command generates a random number. The syntax is:

`random [scale]`

The argument is a whole number from 1 to 225. `random` will display one
random number, between 0 (zero) and the number you specify. For example, the following command will display a random number between 0 and 5:

```plaintext
random 5
```

If you do not specify an argument, `random` will assume a value of 1. Thus, the random number will be either 0 or 1.

Random numbers are often used for sophisticated mathematics. However, random numbers are useful for other, more prosaic purposes. Say Marilyn and Murray are arguing over whose turn it is to cook dinner. All they have to do is enter:

```plaintext
random
```

If the number is 0, Marilyn cooks; if the number is 1, they go to a restaurant.

### The `factor` Command

The `factor` command provides a mathematical service. If you are not interested in math, read this section anyway—you will find it interesting.

If a whole number divides evenly into another whole number, we say that the first number is a FACTOR of the second number. For instance, 4 is a factor of 12. Conversely, we can say that 12 has six factors: 1, 2, 3, 4, 6, and 12.

A number that has no factors except 1 and itself is called a PRIME NUMBER. For example, 14 is not a prime number because it is divisible by 2 and 7; 13 is a prime number. (By definition, 1 is not a prime number.)

The factors of a number that themselves are prime numbers are called PRIME FACTORS. For example, the prime factors of 12 are 2 and 3. Every whole number can be expressed as the product of its prime factors. For example: \(12 = 2 \times 2 \times 3\).

The `factor` command calculates all the prime factors for a number. The syntax is:

```plaintext
factor [number]
```

You can specify any whole number between 1 and 100,000,000,000,000. [Do not use commas when you specify the number.] `factor` will display the number, followed by all its prime factors. For example, if you enter:

```plaintext
factor 12
```

you will see:
If you want to find the prime factors for more than one number, enter the command with no arguments:

```
factor
```

`factor` will read and process numbers, one at a time, until you send an `eof` signal by pressing `<CTRL-d>`.

Short digression:
The seventeenth century Frenchman, Pierre de Fermat, was one of the greatest mathematicians who ever lived. Unfortunately, he had the habit of stating many of his results without writing down the proofs.

Fermat once asserted that all the numbers that followed a particular pattern were prime numbers.

In 1732, the Swiss mathematician Leonhard Euler showed that Fermat was wrong. Euler did this by proving that the fifth such number that followed the pattern, 4,294,967,297, was not prime. In other words, Euler showed that this number has factors other than 1 and itself. This is the only time that anyone has ever shown Fermat to be wrong.

No doubt this took Euler hours of calculation. See how long the same calculation takes Xenix, by entering:

```
factor 4294967297
```

Imagine what Euler and Fermat could have done if they had had Xenix.
CHAPTER 6
TWO USEFUL PROGRAMS

INTRODUCTION
THE bc PROGRAM—A CALCULATOR
THE units PROGRAM—CONVERTING UNITS
INTRODUCTION

There are two valuable programs which come with Xenix that you should know how to use. They are *bc*, which does calculations, and *units*, which converts quantities from one system of measurement to another. This chapter teaches you how to use these programs.

New Words in This Chapter

**VARIABLE**
a quantity with a name that can take on different values

**SCIENTIFIC NOTATION**
a way of writing very large or very small numbers in a compact manner

**EXPONENT**
in scientific notation the number that indicates the scale of the value being described

THE *bc* PROGRAM—A CALCULATOR

Using *bc*

Xenix comes with two built-in programs to perform arithmetic calculations: *bc* and *dc*. They are actually different forms of the same program. *bc*, however, is easier to use, so it is the one that I am going to describe. To start the calculator program, enter the *bc* command. The syntax is:

```
bc
```

You can now enter your calculations, one at a time, by typing an arithmetic expression and pressing <ENTER>. *bc* will perform each calculation as you enter it and display the answer. When you are finished, press <CTRL-d> to stop the program.

The arithmetic expressions that you can enter are straightforward and follow the normal rules of algebra. For example, if you want to add 15545 to 994384,

```
enter: 15545 + 994384
```
The bc Program—A Calculator  57

**bc** will display the answer:

1009929

If you want to do a calculation that involves decimals, you must first use the `scale` command to tell **bc** how many decimal places to use. (Note: this is a **bc** command, not a Xenix command.) The syntax is:

```
scale=number
```

where `number` is the number of decimal places. For example, if you enter:

```
scale=4
```

**bc** will perform all calculations to 4 decimal places of accuracy. If you forget what the setting is, just enter the command name by itself, and **bc** will remind you. For example:

```
scale
```

If you do not set the number of decimal places, **bc** will assume zero and all your numbers will be rounded off to whole numbers. This is a common mistake: People forget to set the number of decimals and then wonder why when they divide 5 by 2 the answer is 2 rather than 2.5.

If you are using an IBM PC AT computer, remember that you can use the numeric keypad to enter your calculations. Press `<NumLock>` in order to enter numbers. Don't forget to press it again when you are finished. As well, the `<*`>, `<->`, and `<+`> keys on the numeric keypad can be used with or without `<NumLock>`.

**Specifying Calculation**

Use the following symbols for calculations:

- `+` addition
- `-` subtraction
- `*` multiplication
- `\` division
- `%` modulo
- `^` exponentiation

Modulo refers to finding the remainder after a division. For example, `36%5` is 1, because 1 is the remainder when 36 is divided by 5. Modulo can only be used with `scale=0`.

Exponentiation means taking a number to a power. For example, `2^3` means $2^3$ (2 to the power of 3); that is, $2 \times 2 \times 2$ or 8. The power must be a
whole number (although it can be negative).

*bc* follows the usual rules of algebra. Multiplication, division, and modulo take precedence over addition and subtraction. Exponentiation takes precedence over everything. As with algebra, you can use parentheses to change the order of operation. For example, if you enter $3+4*5$, the multiplication is done before the addition. If you enter $(3+4)*5$, the addition is done first.

### A Sample Session with *bc*

Here is a sample session with *bc*. The example starts with the shell prompt. The *bc* program is started by entering the *bc* command. Read this section carefully. Make sure that you understand each command that is entered. Notice that unlike the shell, *bc* does not display a prompt.

To make the example easier to read, I will indent each line that is displayed by *bc*.

```plaintext
$ bc
scale = 2
scale
  2
4+3
  7
4-3
  1
4*3
  12
4/3
  1.33
4%3
  1
4 ^ 3
  64
1+3*2
  7
(1+3)*2
  8
<CTRL-d>
$
```

Notice that you use `<CTRL-d>` (the *eof* signal) to stop *bc*. 
Using bc with Variables

You can use bc for more than simple calculations. In particular, you can store intermediate results by using VARIABLES.

A variable is a quantity with a name that can take on different values. With bc, all variables must have a name consisting of one lowercase letter. Thus, you can use up to 26 variables, one for each letter of the alphabet.

The idea is to use variables to store values while you are doing your calculations. To store a value in a variable, use the following syntax:

\[ \text{variable-name} = \text{value} \]

For example, to store the value 5, in the variable a,

```
enter: a = 5
```

Variables are useful to store the results of complicated calculations. For example:

```
b = ((55*23.5)/(344*0.31))^2
```

The value of this calculation is now stored in the variable b.

Once a variable has a value, you can use it in calculations and store the results in other variables.

```
c = b*10
```

```
d = b
```

The first example multiplies the value of b by 10 and stores the result in c. The second example gives d the same value as b.

The equals sign really means "is replaced by." This is not exactly the same as the meaning of the equals sign in algebra. For example, a variable can appear on both sides of the equals sign:

```
b = b+1
```

```
b = 3*b-17
```

In the first example, b takes on the value of itself plus one. This is how you add 1 to a variable. In the second example, b takes on the value of 3 times itself minus 17.

If you want to find out the current value of a variable, simply enter its name. For example:

```
b
```

bc will display the value of the variable.
A Sample Session with bc Using Variables

Here is a session with bc using variables. This example figures out a grocery bill. Someone is buying three avocados at 75 cents each, and five watermelons at $2.15 each. The tax is 2 percent. At the end of the calculations, we will add $1.00 to the total, as a tip for the cashier.

The variables are used as follows:

- \( a \) total price of avocados
- \( w \) total price of watermelons
- \( s \) subtotal
- \( x \) tax
- \( t \) total

Read carefully. Make sure you understand each command that is entered.

```
$ bc
scale = 2
a = 3*0.75
w = 5*2.15
s = a+w
s = 13.00
x = s*0.02
x = .26

```

\( t = s+x \)

\( t = 13.26 \)

\( t = t+1 \)

\( t = 14.26 \)

\(<CTRL-d>\)

$ 

So, the total food cost is $13.26 ($13.00 plus 26 cents tax). With tip, it comes to $14.26.
Using bc for Advanced Calculations

If you are a programmer, or if you are mathematically inclined, you will find that bc can be useful for all kinds of advanced calculations. This section describes these features of bc in technical terms. If you are not interested in this sort of thing, skip to the next section. On the other hand, you may want to read this section anyway, just to see how programmers talk to each other.

bc is a fully programmable mathematical interpreter. It works in extended precision; each number is stored with as many digits as is necessary. As well, you can specify a scale of up to 100 digits to the right of the decimal point. Numbers can be in any base from 2 to 16; conversion is easy.

The programming features of bc allow for function definition, including recursion, in a syntax similar to that of the language C. bc has arrays, local and global variables, and built-in functions. If you write your own functions, you can store them in a file and automatically load them when you start bc.

bc is really a preprocessor for the other calculator program, dc. dc uses a stack and accepts reverse Polish notation. bc is easier to use because it accepts the more familiar infix notation. bc translates each expression into reverse Polish notation and then sends it to dc.

The syntax for the bc command is actually:

```
bc [-cl] [file...]  
```

The -c (compile) option instructs bc to display the reverse Polish expressions, rather than send them to dc. The -l (library) option specifies that you want to use the built-in functions.

If you specify one or more files, bc will read and process them before accepting input from the keyboard. Thus, you can store your own function definitions in a file and load them before you start. For example, if you wanted to use the built-in functions, plus some of your own functions from the files func1 and func2, you would start bc by entering:

```
bc -l func1 func2
```

For a general description of the bc and dc programs, see the Xenix Command Reference manual. For a long, detailed description of bc, see Appendix A of the Xenix Basic Operations Guide.
THE units PROGRAM—CONVERTING UNITS

Scientific Notation

Xenix comes with a program called units that converts from one type of units to another. However, units displays its output as numbers written in a particular format, called SCIENTIFIC NOTATION. The purpose of this section is to teach you how to read numbers that are written in this format, so you can use units. If you already understand scientific notation, skip right to the next section.

Scientific notation is a way of writing down very large or very small numbers in a compact manner. For example, consider the number 3 followed by 18 zeros. One way to write it is:

\[ 3,000,000,000,000,000,000 \]

In scientific notation this would be written as:

\[ 3 \times 10^{18} \]

In other words, \( 3 \times 10 \times 10 \times 10 \times 10 \ldots \text{eighteen times} \).

A very small number, such as 0.000000000004, is represented by:

\[ 4 \times 10^{-11} \]

The minus sign means to divide, rather than multiply. So this number is equal to \( 4 / 10/10/10/10/10/10/10/10/10/10/10/10/10/10/10 \ldots \text{eleven times} \).

The number that indicates the scale of the value is called the EXPONENT. In the first example, the exponent is 18; in the second example, it is -11.

When computers display numbers in scientific notation, they replace the \( \times 10 \) by the letter \( e \) (for exponent). The two numbers above would be displayed as:

\[ 3e+18 \]
\[ 4e-11 \]

units always displays values to seven significant digits. The examples would be displayed as:

\[ 3.000000e+18 \]
\[ 4.000000e-11 \]

If you have never used scientific notation before, it may be a bit confusing at the beginning. If so, just think of it like this: The exponent (the number after the \( e \)) tells you where to place the decimal point. If the
exponent is positive, move the decimal point that many places to the right; if the exponent is negative, move the decimal point to the left.

Here are some examples. On the left is a number, in scientific notation, the way `units` would display it. On the right is the same number the way you would normally write it.

<table>
<thead>
<tr>
<th>Scientific Notation</th>
<th>Normal Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.456789e+02</td>
<td>345.6789</td>
</tr>
<tr>
<td>6.543210e-03</td>
<td>0.006543210</td>
</tr>
<tr>
<td>3.000000e+18</td>
<td>3000000000000000000000000</td>
</tr>
<tr>
<td>4.000000e-11</td>
<td>0.00000000004</td>
</tr>
<tr>
<td>2.468024e+00</td>
<td>2.468024</td>
</tr>
</tbody>
</table>

Notice in the last example that the decimal point doesn’t move because the exponent is zero.

**Using units**

The `units` program helps you convert quantities from one system of measurement to another. To start the program, enter the `units` command. The syntax is:

```
units
```

`units` will ask you to enter what you have and what you want. For example, if you want to convert yards to inches, you “have” yards and you “want” inches. `units` will display:

```
you have: yard
```

and you will enter:

```
yard
```

`units` will then display:

```
you want: inch
```

and you will enter:

```
in
```

`units` will then display two conversion factors. The display will look like this:

```
you have: yard
you want: inch
  * 3.600000e+01
  / 2.777778e-02
```
Notice that the numbers are in scientific notation. Their values are 36 and 0.02777778.

The first number is what you would have to multiply yards by to get inches. The second number is what you would have to divide yards by to get inches. (The * means multiplication, and the / means division.) In other words, 1 yard equals 36 inches, and 1 inch equals 0.02777778 yards. Most of the time you can ignore the second number.

units can also convert a specific number of units. For example, if you wanted to convert 2 yards to inches, it would look like this:

you have: 2 yard
you want: inch
* 7.200000e+01
/ 1.388889e-02

In this case, 2 yards equal 72 inches. (Ignore the second number.) Notice that you do not have to specify a plural form of what you have or want. In other words, you do not need to specify "yards"; "yard" is fine.

One more example: This time to convert 100 miles/hour to centimeters/week.

you have: 100 miles/hour
you want: cm/week
* 2.703698e+09
/ 3.698638e-10

Thus, 100 miles/hour equals 2,703,698,000 centimeters/week.

The Capabilities of units

units will recognize just about any units that you can think of: scientific, mathematical, weights and measures, and so on, including both the metric and British systems. Not only that, but you can use any reasonable abbreviation. However, by its nature, units can only convert quantities by multiplication. Thus, it cannot convert from centigrade to Fahrenheit.

Altogether, units recognizes over 400 different units and abbreviations. If you come across any sort of strange unit, even if it is outdated, try it; it's probably in there.

All of the abbreviations and conversion factors are stored in a file by the name of /usr/lib/unittab. Later in the book, you will learn how to display files. At that time, you can display this file and take a look at all your choices.
Communicating with units

If you make a mistake, units will let you know in one of two ways. If it cannot understand a word, it will display the message:

\texttt{can't recognize}

followed by the word. And if you try to convert incompatible units (say, inches to hours), units will display the cryptic message:

\texttt{conformability}

meaning that it can't make the first unit conform to the second.

When you are using a unit that consists of more than one word, type it as one long word without spaces. For example, for “light year” use \texttt{lightyear}. If you want British units, preface the name with “br”. For example, \texttt{brgallon}.

For the convenience of users who do scientific calculations, units recognizes important constants. The most useful are:

\begin{itemize}
  \item \texttt{pi} ratio of circumference to diameter of a circle
  \item \texttt{c} speed of light
  \item \texttt{e} charge on an electron
  \item \texttt{g} acceleration due to gravity
  \item \texttt{mole} Avogadro's number
  \item \texttt{au} astronomical unit
\end{itemize}

Use these names anywhere you would use a number. For example:

\begin{verbatim}
you have:  c
you want:  mi/sec
  \ast  1.862824e+05
  /  5.368193e-06
\end{verbatim}

\begin{verbatim}
you have:  c
you want:  km/sec
  \ast  2.997925e+05
  /  3.335640e-06
\end{verbatim}

Thus, the speed of light is approximately 186,282.4 miles/second, or 299,792.5 kilometers/second.

Converting Money

As I mentioned above, there are some types of conversion that units cannot do (for example, centigrade to Fahrenheit). However—possibly
by way of compensation—units will convert money. Use the name of the country, followed by the name of the currency. For example:

\begin{verbatim}
  you have:  10 germanymark
  you want: japanyen
  * 1.176324e+03
  / 8.501061e-04
\end{verbatim}

Thus, 10 German marks equals \(1,176.324\) Japanese yen.

units recognizes some currencies without the country name. For example, yen (japanyen), pound (englandpound), franc (francefranc), and so on. (Thus, if you want to use “pound” meaning weight, you must specify lb.) You can refer to the American dollar as both dollar and buck.

The only problem with the money conversion system is that the values were taken from the Wall Street Journal on May 18, 1977. However, later in the book, you will learn how to edit files. At that time, you will be able to change the file with the conversion factors, /usr/lib/unittab, to reflect more current values (if you are the system manager).
CHAPTER 7

UNDERSTANDING THE XENIX FILE SYSTEM

INTRODUCTION
UNDERSTANDING FILES
HOW IS INFORMATION STORED?
HOW DOES XENIX HANDLE FILES?
WORKING WITH A FILE SYSTEM
INTRODUCTION

Xenix has a carefully designed system to access data. The purpose of this chapter is to introduce you to this system. You will start by learning how information is represented in the computer and go on to learn how Xenix organizes this information into workable units.

New Words in This Chapter

FILE
a source from which data can be read or a target to which data can be written

ORDINARY FILE
a file that contains data

DIRECTORY
a file that is used to keep track of other files

SPECIAL FILE
a file that represents a physical device

BIT
the smallest component of computer memory

BYTE
a sequence of eight bits

DECIMAL SYSTEM
a system of counting using the digits 0 through 9

BINARY SYSTEM
a system of counting using only the digits 0 and 1

ASCII CODE
a standardized representation of characters by particular patterns of bits

TEXT FILE
an ordinary file in which each byte represents one character

BINARY FILE
an ordinary file in which the bytes are considered to represent numbers in the binary system
SOURCE PROGRAM
a program in a form that people can understand

EXECUTABLE PROGRAM
a program stored in a form that the computer can understand

EDITOR
a program that you use to create and modify text files

INUMBER
“index number”; a unique number, used internally by Xenix to represent a particular file; pronounced: “eye-number”

LINK
an entry in a directory for a particular file

OWNER
the userid that controls the access to a particular file

PARENT DIRECTORY
a directory that contains another directory

SUB-DIRECTORY
a directory that is contained in another directory

TREE-STRUCTURE
a scheme of organizing files in which the hierarchy resembles the outline of a tree

ROOT DIRECTORY
the main directory of a tree-structured file system

PATH
a sequence of directories that leads through the tree to a particular directory

PATHNAME
the full specification of a file, including the path to the directory in which the file lies, followed by the name of the file

WORKING DIRECTORY
the name of a particular directory from which you can specify paths

HOME DIRECTORY
the directory to which the shell automatically sets your working directory whenever you log in
UNDERSTANDING FILES

What Is a File?

In English, the word "file" refers to a collection of papers or documents. A salesperson might keep a file of customers. As a computer term, "file" usually refers to a collection of data that is given a name and saved on some type of storage device. For example, a programmer might keep a number of files on a diskette. The files may be programs, documents, and so on.

In Xenix, the term FILE has a more general meaning. It refers to any source from which data can be read, or any target to which data can be written. For example, to print data, you write the data to the file that represents the printer. To receive data from the keyboard, you read data from the file that represents the keyboard.

In Xenix, all I/O (input/output) is done via files. As far as the user is concerned, Xenix makes only a minimal distinction between a file that holds a document and a file that represents a physical device like a printer.

This generality is of enormous importance. It makes all the stored data and all the communication devices available in the same simple, standard manner. For instance, when you run a program, you can choose where you want the output sent: to the printer, to the display, to a file on disk, and so on. Thus, it is easy to have any program print, display, or save its output. (This is all discussed later.)

The Three Types of Files

There are three types of files: ORDINARY FILES, DIRECTORIES, and SPECIAL FILES.

Ordinary files contain data: programs, documents, and so on. These are the files that you work with most of the time.

Directories are used to organize all the files into a convenient, easy-to-use system. As the name implies, each directory keeps track of a number of files. The idea is to organize your files into directories in a way that suits your work.

Special files represent physical devices. To read or write to a device, you simply read or write to the special file that represents that
device; Xenix takes care of all the details. The various special files can be used to access the console (display and keyboard), the printer, remote terminals, the built-in clock, the disks, and even the memory. (Special files are described in more detail in Chapter 15.)

Sometimes when people say “file” they mean any file, and sometimes they mean an ordinary file. In this book, I will be specific if the meaning is not obvious from context.

HOW IS INFORMATION STORED?

Bits and Bytes

Xenix handles all the details of file storage automatically. However, there are a few basic concepts that you should understand.

Computer memory consists of a large number of tiny components, called BITS. At all times, each bit has one of two possible values. These values are usually referred to as 0 (zero) and 1 (one).

Information is stored in the computer memory as sequences of bits. And we can represent those sequences of bits as 0’s and 1’s. For example, a particular small section of memory might be represented as 00011001.

The memory is organized so that the bits are grouped together in sequences of eight. Each of these sequences of eight bits is called a BYTE. It is customary to measure computer memory in bytes.

Where Does the Word “Bit” Come From?

When we count, we use a system called the DECIMAL SYSTEM based on the digits 0 through 9. Mathematicians have developed another system, based on only the digits 0 and 1, called the BINARY SYSTEM. In the decimal system (our usual system), each number is made up of the digits 0 through 9. In the binary system, each number is made up of only the digits 0 and 1. For example, 00011001.

A byte is often considered to represent an eight-digit binary number, each bit being one of the digits. Hence, the name “bit,” which stands for “binary digit.”
How Is Data Represented? The ASCII Code

If the computer memory works in bits and bytes, how does it store information that humans use—names, numbers, and so on? The answer is that a code is used.

Each byte consists of a sequence of eight 0's and 1's. If you were to figure it out, you would see that there are exactly 256 possible patterns. In other words, each byte must have one of 256 possible values. These values are referred to by number, from 0 to 255. (With computers, it is often more convenient to begin counting with zero rather than with one.)

The code that is used is called the ASCII CODE. ("ASCII" stands for "American Standard Code for Information Interchange.") The ASCII code uses 128 of the 256 possible patterns. Each of the patterns corresponds to a particular character. For example, the code for the letter A is 01000001. Since lower-case letters are considered to be different from upper-case letters, they have their own codes. The code for a is 01100001.

Of course, the upper- and lower-case letters require only 52 different patterns. Some of the other patterns are used for the digits (0 through 9), and the special characters (+, $, #, &, and so on). The rest are used for characters such as space, newline, and tab; and the special signals kill, Intr, erase, eof, start, and stop.

The computer stores information that consists of letters, numbers, and special characters as a series of bytes. Each byte contains the pattern for one character. For example, a file that holds 100 characters is 100 bytes long.

Two Types of Ordinary Files: Text Files and Binary Files

There are two types of ordinary files. Files in which each byte represents one character are called TEXT FILES. Most of the files that you create will be text files: documents, programs, tables of information, and so on.

A text file is stored as a long sequence of bytes, each byte representing one character of data. If the information in the file consists of lines, then the end of each line is marked by a newline character.

However, there are some ordinary files in which the bytes are not interpreted as characters. Rather, each byte is considered to be a
number in the binary system. This type of ordinary file is called a
BINARY FILE. Binary files have one main use.

If you were to write a program, it would be stored as a text file. The
instructions in the program would be composed of characters, each
stored as one byte.

However, before a program can be executed (most of the time), it
must be processed in a certain way. The instructions in the program
have to be translated into instructions that the computer can execute.
The original program is called a SOURCE PROGRAM and the translated
program is called an EXECUTABLE PROGRAM.

The details are not important right now. What is important is that
while the source program (that you can understand) is stored as a text
file, the executable program (that the computer can understand) is
stored as a binary file.

If you want to see what is in a text file, you can display it. As Xenix
reads the file, it displays the characters that correspond to the bytes of
the file. Whenever Xenix encounters a newline character, it starts a new
line.

If you were to display a binary file, it would come out as gibberish
because the bytes are meant to be understood only by the computer;
they do not correspond to characters.

Obviously, the files that you create from the keyboard will all be
text files. Will you use any binary files?

The answer is yes. First of all, you may write your own programs. If
you do, you will have them translated into executable programs that are
stored as binary files. More important, however, is that every time you
enter a command you use a binary file.

Remember that commands are processed by the shell. When you
enter a command, the shell processes it by executing a program whose
job it is to perform that command. For example, when you enter the date
command, the shell executes a program called date that performs this
command.

Thus, Xenix comes with many programs, one for each command.
And each of these programs is stored as a binary file. (Actually, a few
commands are handled directly by Xenix, but we can ignore that for
now.) So every time you enter a command, you are using a binary file.

So, in summary, there are two types of ordinary files. Text files
contain information in the form of characters. Most of your files will be
text files. Binary files hold executable programs. Almost all the Xenix
commands are performed by separate programs, each of which is stored
as a binary file.
HOW DOES XENIX HANDLE FILES?

File Names

Every file has a name. A file name can be up to 14 characters. When you give a name to a new file, you can use any characters you want including upper- and lower-case letters and numbers. Although you can use special characters, most of them have special uses. To be safe, do not use any special characters except a period (.).

Here are some examples of valid file names:

```
    test  Test number7  84.1  data.old  data.new  .profile
```

Remember, Xenix considers upper- and lower-case letters to be different, so `test` and `Test` are two different names. As a general rule, Xenix names are lower case. If you ever encounter a problem where Xenix can't seem to find a file, make sure that you have not inadvertently entered a lower-case letter as upper case.

There is a particular use for file names that begin with a period (such as `.profile`). There are two commands to list the names of files in a directory, `ls` and `lc`. Unless you specifically request it, these commands will not list file names that begin with a period. This feature is used only for certain files with special uses. (`.profile` is such a file. You will learn about it later.)

Where Do Files Come From?

Xenix automatically creates a file whenever one is needed. For example, say you enter the command to make a copy of a file. The copy is to be called `newfile`. If `newfile` does not already exist, Xenix will create it.

As you put more data into a file, Xenix automatically extends the file. When you remove data from a file, Xenix shrinks the file. When you are finished with a file, all you need to do is tell Xenix to remove the file from its directory.

The bottom line is that Xenix maintains all your files automatically. You never need to ask for a file to be created; nor do you need to concern yourself with any of the details of managing your files.

Usually, you will make new files by using a program called an EDITOR. An editor allows you to create and modify text files. There are several editors that come with Xenix.
several editors that come with Xenix.

When you want to make a new file, you execute the editor and tell it the name of the file. The editor will call upon Xenix to create the new file for you. Using the editor, you enter information from the keyboard to go into the new file. From then on, you use the editor any time you want to modify or add to the file.

What Is in a Directory?

Xenix maintains directories to allow you to organize your files. A directory contains the names of a number of files. You can make new directories or remove old ones, as you see fit. The idea is that you can make a directory for every group of files that you use. For example, you might have three directories: one for programs, one for correspondence, and one for customer information files.

In order to keep track of all the files, Xenix assigns each file a unique identification number, called an INUMBER. ("inumber" stands for "index number"; it is pronounced: "eye-number.")

We usually speak of a directory as containing a number of files. However, directories do not really contain files: They contain entries for files. Each entry consists of a file name and an inumber. Xenix stores the actual file somewhere else.

When a directory holds an entry for a particular file, we say that there is a LINK from the directory to that file. Most of the time, it is convenient to refer to a file as being in this or that directory. What we really mean is that the directory contains a link to the file.

Whenever you use a file, all you need to tell Xenix is the name of the file and the directory in which it appears. Xenix automatically looks up the inumber and accesses the file. Because Xenix handles all the linking, you only need to know the file name, not the inumber.

You might be wondering why the Xenix file system was designed this way. One reason is that linking makes it easy for people to share files. It is a simple matter for more than one directory to contain file names that link to the same file. The file names do not even have to be the same.

Thus, one user can have a file in his or her directory under a particular name. At the same time, another user can have the same file in a directory under a different name. And since the directories only contain links, Xenix only has to keep one copy of the file.
Owning Files

If a file can appear in more than one directory, how does Xenix coordinate the different links? What if one user creates a file and another user links the file into the directory? Xenix considers all links to be equal. Does this mean that a user can access and modify anybody’s files?

The answer is that Xenix considers each file to have an OWNER. The owner is the userid that controls the access to the file. Although anyone can link to a file (that is, place it in one of his or her directories), the file cannot be used unless the owner has given permission. (The details are explained in Chapter 17.)

Note: Users do not own files—userids own files. This makes sense: Xenix does not know about users (people), it only knows about userids (internal names). Obviously, if you give someone your password and that person logs in under your userid, he or she will have control of your files. Thus, when I refer to “your” files, I mean the files that are owned by your userid.

Who Owns the System Files?

Xenix comes with a great many files that are used to run the system. Since all files must be owned by some userid, it seems fair to ask, Who owns the system files?

Conceptually, Xenix is a universe, populated by userids, files, and processes. (Processes, discussed later, are programs that are executing.) Inside Xenix, people do not exist. It is the userids, not the users, that log in and out, own files, and start processes. Although you may think of yourself as a person, Xenix only knows you as one of its userids.

If you were to ask someone at random—Plato, for instance—he might tell you that userids exist only because each one represents a real person. However, there are four userids that do not correspond to real people. These userids are: root, bin, sysinfo, and uucp.

Each system file is owned by one of these four userids. bin owns most of them, the ones that contain programs to perform the various commands. These are all binary files, hence the name bin, which stands for “binary.”

root, the userid of the superuser, owns a few files that are used for superuser functions. sysinfo, which stands for “system information,” owns a few files that contain important system data. uucp stands for
"Unix to Unix copy." It is the name of a program that transfers data between two Xenix or Unix systems, and it owns some files needed for this purpose.

WORKING WITH A FILE SYSTEM

Organizing with Sub-Directories

A directory can contain entries for any type of file. This means that a directory can contain not only ordinary files and special files but also other directories. When a directory contains a second directory, we call the first one the PARENT DIRECTORY, and the second one a SUB-DIRECTORY. The advantage of sub-directories is that they allow you to group your files into categories and sub-categories. Here is an example showing how sub-directories can be used to organize files.

It is your job to organize a file system for the Acme Machine Company. You need a file for employee records and a file for payroll records. As well, you need two files for each salesperson and two files for each repair person. The sales files keep track of customers and supplies. The repair files keep track of machines and [different] supplies. There are two sales people—Dick and Jane—and three repair people—Curly, Larry, and Moe.

Here is how you might organize the file system: Put everything under one directory called acme. acme contains two files and two sub-directories. The files are employees and payroll. They hold the records for the employees and payroll, respectively. The sub-directories are sales and repairs. They hold the files for the sales and repair people.

The sales directory has two sub-directories: dick and jane. Each of these sub-directories contains two files: customers and supplies. The repairs directory has three sub-directories: curly, larry, and moe. Each of these sub-directories contains two files: machines and supplies.

Take a look at Figure 7.1, which shows the structure of this system. Notice how the directories serve to organize the files. Although our example uses only a handful of files, sub-directories allow you to organize as many files as you need.

Not only is this system easy to understand, it is easy to maintain. For example, whenever a sales or repair person leaves the company, all we need to do is remove his or her directory. When a new person is recruited, we make a new directory.
This diagram shows the organization of files under the directory `acme`. The details are described in the text.
Remember that a file is known by its name and its directory. There is no problem having two files with the same name (supplies) as long as they are in different directories (sales and repairs). In the same way that you would not confuse Robert Redford with Robert Prior, Xenix recognizes the two files as being different, because they are in separate directories.

**The Tree-Structured File System**

Take another look at Figure 7.1. Notice that the directories form a hierarchy, with **acme** at the top. This sort of organization is sometimes called a TREE-STRUCTURE. You can imagine the diagram as showing the outline of an upside-down tree. Think of each link between a parent and sub-directory as a branch, and each ordinary or special file as a leaf. The main directory (in this case, **acme**) is called the ROOT DIRECTORY of the tree. Xenix organizes its files into a large tree-structured file system.

As you remember, it is the job of the system manager to register new users. One of the things that Xenix does for a new user is to make a new directory for him or her in the **usr** directory.

All of your files and sub-directories will be contained in or under the directory with the same name as your userid. They will form a sub-tree, attached to the Xenix tree under the **usr** directory. For example, if your userid is **dianne**, all your files will be under the directory **dianne**, which is under the directory **usr**.

Whenever you log in, Xenix sets things up so that you are automatically working in the directory that serves as your main directory. As you work, you can move from directory to directory as the need arises. (This is all explained in Chapter 8.)

Take a look at Figure 7.2. This figure shows an outline of a Xenix file system. In this system, there are three users. Their userids are **acme**, **harley**, and **dianne**. Each userid has its own directory under the **usr** directory.

Notice that the root directory has the curious name of /. This is the directory from which the entire Xenix file system emanates. One of the reasons that the name is written as / is because it is used a lot, and it is much easier to type / than it is to type “root.”

Now you can understand why the userid of the superuser is called **root**. It is named after the root directory—the main stem of the whole file system.
Figure 7.2 AN OUTLINE OF THE XENIX FILE SYSTEM

This diagram shows an outline of the Xenix file system. There are eight subdirectories under the root directory. All the files that belong to users are under the usr sub-directory. In this example, there are three userids: acme, harley, and dianne.

Specifying a File—Pathnames

To access a file, Xenix needs to know the name of the file and the directory in which it lies, so you need to know how to specify this information.

If you think about it, you will see that, starting from the root directory, there is exactly one way to get to any particular directory. The sequence of directories that you pass through is called a PATH.

Consider the files shown in Figures 7.1 and 7.2. Say that the tree-structure in Figure 7.1 is attached to the main tree in Figure 7.2. To get to the sales directory, you start from the / (root) directory and pass
through the \texttt{usr} directory and the \texttt{acme} directory. To specify this for Xenix, we would write the names of the directories, in order, separated by / characters.

\texttt{/usr/acme/sales}

The first / stands for the root directory. You pronounce it as “slash.” Thus, you would pronounce \texttt{/usr/acme/sales} as “slash-usr-slash-acme-slash-sales.”

To specify a particular file, you use the path to the appropriate directory, followed by /, followed by the file name. For example, here is how to specify the \texttt{supplies} file in the \texttt{sales} directory:

\texttt{/usr/acme/sales/supplies}

Here is how to specify the \texttt{supplies} file in the \texttt{repairs} directory:

\texttt{/usr/acme/repairs/supplies}

Now you see why there is no chance that Xenix will confuse the two files, even though they have the same name.

The full specification of a file—the path to the appropriate directory, followed by the file name—is called a PATHNAME. Here are two more examples of pathnames: First, Xenix contains a file named \texttt{boot} in the / directory:

\texttt{/boot}

Second, Xenix contains a file named \texttt{ascil} in the /\texttt{etc/default} directory:

\texttt{/etc/default/ascil}

If you misspell the name of a file, Xenix will tell you that it can’t find that file. Remember that Xenix distinguishes between upper and lower case. Almost all the time, you will be using lower case. If you use an upper-case letter in the wrong place, Xenix will consider it to be a different name. For example, if you specify:

\texttt{/etc/default/Ascil}

Xenix will tell you that it cannot find the file \texttt{Ascil}. Whenever you see a message like this, check your spelling.

\textbf{Your Working Directory}

You can always refer to a file by specifying its pathname. However, most of the time you will be working in one directory, and it is irritating to have to specify the path to that directory over and over.
For example, say that you want to make a copy of the file `supplies` in the directory `/usr/acme/sales`. The name of the new file will be called `newfile` and will be in the same directory. You can use the `cp` command by entering:

```
    cp /usr/acme/sales/supplies /usr/acme/sales/newfile
```

Obviously, there must be a better way. In fact, there is a second way to specify files, using what is called your WORKING DIRECTORY.

At all times, Xenix remembers the name of one of your directories. This is your working directory. You can specify a file in two ways. First, you can use a pathname that starts from `/` (the root directory); or, you can use a pathname that starts from your working directory.

When you use a pathname that begins with `/`, Xenix assumes that the path starts from the root directory. When you use a pathname that does not begin with `/`, Xenix assumes that the path starts from your working directory.

For example, if your working directory is `/usr/acme`, the following two commands are equivalent:

```
    cp /usr/acme/sales/supplies /usr/acme/sales/newfile
    cp sales/supplies sales/newfile
```

If your working directory were `/usr/acme/sales`, the command could be even shorter:

```
    cp supplies newfile
```

You can change your working directory to another directory any time you want. (See Chapter 8.) You can even change to a directory that contains files that are not owned by your userid. The idea is to set your working directory to the one in which you will be doing most of your work—this makes most of your pathnames as short as possible. If you ever need to use a file in another part of the tree, you can specify the full pathname.

You might wonder whether you have to set your working directory every time you log in. The answer is no: The shell sets it for you. Every userid has a HOME DIRECTORY. Whenever you log in, the shell automatically sets your working directory to be your home directory.

The home directory is the one in the `/usr` directory that has the same name as your userid. For instance, if your userid is `acme`, your home directory is `/usr/acme`. When you log in, the shell will automatically set your working directory to `/usr/acme`. As you work, you can change your working directory as you see fit.
INTRODUCTION
DISPLAYING INFORMATION FROM TEXT FILES
WORKING WITH DIRECTORIES
DISPLAYING INFORMATION FROM DIRECTORIES
PRACTICING THE DIRECTORY COMMANDS
INTRODUCTION

In the last chapter, you learned about the Xenix file system. In this chapter, you will learn the practical details that you need to work with the system. In particular, you will learn how to display the information in text files and directories and how to work with directories. The chapter ends with a short practice session. I strongly advise you to log in and go through the session on your own terminal.

New Words in This Chapter

SCROLL
to display a line of information on the display screen by writing to the bottom of the screen and moving all the other lines up one position

DISPLAYING INFORMATION FROM TEXT FILES

The more Program—Displaying a File

Xenix offers many different ways to display files. By far the best is the more program. Unfortunately, this program should have been named "display," rather than "more." Don’t let the name confuse you: This is the way to display files.

more displays the information in the file one screenful at a time. If the file is small enough, more will display all of it at once. If not, more will display the first screenful and pause. You can either ask for the next screenful or enter one of several commands.

more was written at the University of California at Berkeley. Like many Berkeley programs, it comes with a large number of options and commands. In this section, I will describe only the most useful aspects of the program. If you are interested in the details, check the Xenix Command Reference manual.

Before you continue reading, log in so you can follow along with the examples as you read.

Using the more Program

To start the more program, enter the more command. The syntax is:

more [-cs] file
The two options will be explained below. The argument is the name of the file that you want to display. Usually, you will not use any options. For example, to display the file `myfile`,

```
enter: more myfile
```

As I explained in Chapter 7, there are three types of files: special files, directories, and ordinary files. There are two kinds of ordinary files: binary files and text files.

Text files are the ones that contain information in a form that people can understand. The `more` program will only display text files. (There are other commands to display directories.) If you try to display another type of file, `more` will tell you that it is not a text file and ignore your request.

For practice, enter the command to display the file `/usr/lib/unittab`. This is the file that contains the information used by the `units` program. (See Chapter 6.)

(To review, the name `/usr/lib/unittab` means that the file name is `unittab`, and the file is in the `lib` directory, which is in the `usr` directory, which is in the root directory.)

Enter the command:

```
more /usr/lib/unittab
```

You will see the first screenful of information. At the bottom of the screen, you will see:

```
--more--(4%)
```

This shows the percentage of characters in the file that have been displayed so far; in this case, 4 percent. The message `--more--` means that there is more to display. (As strange as it may seem, this is how the name of the program was chosen.)

You can now use any of the `more` commands to display the file as you wish. With most commands, it is not necessary to press `<ENTER>`. For example, to use the d command, just press `<d>`.

Most of the time, you will use only three commands. They are:

```
<SPACE> display the next screenful
<d> display the next half screenful
<ENTER> display the next line
```

Take a moment and try each of these commands a few times.

If you want to go back to the beginning of the file, press `::<p>`. `more` will skip to the beginning of the file. You can then press `<SPACE>` to redisplay the first screenful of information.

```
::<p> skip to the beginning of the file
```
Try this command now, and then press <SPACE> a few times. 
more stops automatically as soon as it has displayed the last line of the file. If you want to stop before the entire file has been displayed, press <DEL>.

<DEL> stop the program

Try it now. Notice that the program stops immediately and you see the shell prompt.

The more Program—Searching for Patterns

Before we continue, enter the more command again:

more /usr/lib/unittab

more has a command to allow you to search the file for a particular pattern. Press </> (slash) followed by the pattern, and then press <ENTER>. more will skip to the next line that contains that pattern. If more cannot find the pattern, it will tell you. For your convenience, more displays two lines before the required line.

</>pattern<ENTER> skip to the next line containing pattern

Try this example:

</>gallon<ENTER>

Notice that the / and the pattern are displayed on the bottom line as you type. This allows you to press <BACKSPACE> to correct a mistake before you press <ENTER>.

As soon as you press <ENTER>, more skips to the next line that contains gallon. Now, skip to the line that contains the string dollar. Type:

</>dollar<ENTER>

more allows you to be even more specific. If you type a number before the /, more will skip that number of times. For example, if you type:

4/dollar<ENTER>

more will skip to the fourth occurrence of the pattern dollar (starting from the current position). Try it.

Here is a summary of the pattern matching command:
At any time, you can repeat the last command by pressing `<.>` (the period). A common use of this is to search for a pattern and then keep pressing `<.>` to search for the pattern repeatedly.

`<.>` (period) repeat last command

**The more Program—Advanced Commands**

In this section, I will describe some of the advanced commands for the `more` program. If you want, you can skip this section and come back to it later when you have more experience.

First, you can press `<?>` to display a list of all the possible commands. The command list takes up the entire screen. When you are finished, you can either type another command, or you can press `<CTRL-l>` to redisplay the current area of the file. (Note: This is the letter `l`.)

`<?>` display command list
`<CTRL-l>` display current area of file

Second, you can temporarily stop `more` and enter a Xenix command. To do this, press `<!!>`, followed by the command and press `<ENTER>`.

`<!!>command` enter a Xenix command

For example, if you are in the middle of a file, and you need to know what time it is, type:

`<!!>date<ENTER>`

If you want to enter a few commands, you can start a new copy of the shell from within `more` by typing:

`<!!>sh<ENTER>`

You can now enter as many commands as you want. When you are finished, press `<CTRL-d>`. This will end the new copy of the shell, and you will be back in `more`, exactly where you left off. If you are using the C-Shell,

enter: `<!!>csh<ENTER>`

In Chapter 10, I will show you how to use the `vi` program to edit files. You can switch to the `vi` program from the middle of `more` by pressing `<v>`. When you stop `vi`, you will be back in `more`. You may have to press `<CTRL-l>` to display the current area of the file.
<v> start the vi editor program

Notice that more displays a file by writing each new line at the bottom of the screen and moving all the other lines up one position. This is called SCROLLING. Even though most terminals scroll quickly, it is not pleasing to the eye when a whole screenful of information is changed in this manner.

You can control how more displays information by using two options when you start the program. First, you can use the -c option to specify that whenever a new screenful of information is to be displayed, more will redraw the entire screen, rather than scroll line by line. Second, you can use the -s option to specify that wherever there is more than one consecutive blank line, more will “squeeze” them and display only a single blank line.

Summary of the more Command

The more command has the following syntax:

```
more [-cs] file
```

The -c option instructs more to redraw each screenful of new information, rather than scrolling line by line. The -s option instructs more to “squeeze” out consecutive blank lines.

more has a number of commands that you can use while it is displaying a file. Most of the commands do not require you to press <ENTER> after you type them.

- <SPACE> display the next screenful
- <d> display the next half screenful
- <ENTER> display the next line
- <:p> skip to the beginning of the file
- <DEL> stop the more program
- [n]<\>/pattern<ENTER> skip to nth line that contains pattern
- <.> (period) repeat last command
- <?> display command list
- <CTRL-!> display current area of file
- <!>command enter a Xenix command
- <v> start the vi editor program
For a more extensive description of the *more* program, see the Xenix Command Reference manual.

**The head and tail Commands**

If you want to look at a few lines at the beginning or end of a file, it is faster to use the `head` and `tail` commands rather than the `more` program. The `head` command displays lines at the beginning of a file. The `tail` command displays lines at the end of a file. The syntax is:

```
head [-number] file
tail  [-number] file
```

When you enter these commands, you must specify the name of the file that you wish to display. If you specify a number as an option, `head` and `tail` will display that number of lines. For example, to display the first 15 lines of the file `/usr/lib/unittab`,

```
head -15 /usr/lib/unittab
```

If you don’t specify a number, `head` and `tail` will display ten lines. For example, the command:

```
tail /usr/lib/unittab
```

displays the last ten lines of the file.

**WORKING WITH DIRECTORIES**

**How to Think about Directories**

In Chapter 7, I explained how Xenix uses directories and sub-directories to organize a tree-structured file system. At any time, one of the directories in the tree is designated as your working directory. If a file is in your working directory, it is easier to use because you don’t have to specify a pathname.

For example, if your working directory is `/usr/harley/book` and you want to work with the file `/usr/harley/book/chapter.2`, you can refer to it as `chapter.2`.

Xenix has a number of commands specifically designed to let you work with the file system. The most important are `pwd`, `cd`, `ls`, and `ls`. 
The **pwd** command displays the name of your working directory. The **cd** command lets you change your working directory. The **lc** and **ls** commands display information about the files in a directory.

As you work, it is helpful to keep an image of the tree-structure in mind. Think of yourself as sitting in a tree that represents the file system. Each branch of the tree has one box on it. This box is a directory. Inside the box are the names of files. At any given time, you are sitting in one of the boxes. This box is your working directory. When you tell Xenix the name of a file, Xenix assumes that unless you specify differently, the file is in the same box as you.

So, the **pwd** command tells you the name of the box in which you are sitting. The **cd** command lets you jump to a box on another branch. And the **lc** and **ls** commands let you look inside any box you want.

The idea is to use the **cd** command to position yourself in the box that contains most of the files that you will be using. Thus, at any time, you can think of yourself as being “in” your working directory.

### The **pwd** Command

The **pwd** command tells you what directory you are in. In other words, the **pwd** command displays the name of your working directory. The syntax is:

```
pwd
```

The name **pwd** is an abbreviation for “print working directory.” (Remember, the Xenix tradition is to use the term “print” to mean “display.”)

Whenever you log in, the shell will set your working directory to your home directory. Remember, your home directory, which doesn’t change, is where you start each time you log in. Your working directory, which you can change, is where you are at a particular time. Some people enter the **pwd** command as soon as they log in to remind themselves where they are.

### The **cd** Command

The **cd** command puts you in a different directory. In other words, the **cd** command changes your working directory. The syntax is:

```
cd [pathname]
```
This command changes your working directory to the directory specified by the pathname. For example, to change your working directory to /usr/harley/book,

    cd /usr/harley/book

If you don't specify a pathname, cd will change to your home directory. For example:

    cd

In order to control access to the various files and directories in the system, Xenix maintains a system of permissions. Permissions are explained in Chapter 17. For now, just understand that you can only access those files and directories for which you have permission.

When you use the cd command, you can specify a pathname to any directory in the whole file system, even one that does not belong to your userid. The cd command will check to make sure that you have permission to examine the directory. If you do, cd will go ahead with the change. If not, cd will display the pathname you specified followed by:

    :bad directory

One nice thing is that the Xenix file system is set up so that you have permission to examine (but not modify) just about every directory in the system. This means that you can use the cd command to zip around and explore every nook and cranny of the tree.

DISPLAYING INFORMATION FROM DIRECTORIES

Introduction to the lc and ls Commands

The lc and ls commands display information about files and directories. The commands are similar. In fact, lc was designed at U.C. Berkeley as a newer version of ls. The name lc is an abbreviation for “list in columns”; ls is an abbreviation for “list.”

The best idea is to learn lc first and use it almost all the time. Learn ls as an alternative, but only for one particular case in which it happens to be faster.

The lc command has 22 options, and the ls command has 12. As usual, I will describe only the important options. If you are interested in the others, check the Xenix Command Reference manual.
In the next few sections, I will describe these commands, using only short examples. Following this, you will find a section of more extensive examples, tying together the `pwd`, `cd`, `lc`, and `ls` commands.

The `lc` Command

The `lc` command displays information about files and directories. This section describes how to use the `lc` command with its most common options. The next section describes a few of the less common options.

The syntax for the `lc` command is:

```
lc [-1adiFR] [name...]  
```

(Notice that the first option in the list is the number 1, and the fourth option is the small letter l. Also, notice that the last two options are upper-case letters.)

You can specify one or more names, which can be either directories or files. For example:

```
lc book  
```

If you specify the name of an ordinary or special file, `lc` will display information about that file. If you specify the name of a directory, `lc` will display information about all the files in that directory. If you do not specify a name, `lc` will display information about the files in your working directory. For example:

```
lc  
```

When you use `lc` without options, it displays only file names. So if you enter the command above, you will see the names of all the files in the working directory.

When `lc` displays file names, it normally groups them into columns. The `-1` option (the number 1) displays each name on a separate line. For example:

```
lc -1 book  
```

The `-1` option (the letter l) displays more information than just the file name. (l stands for “long.”) Specifically, this options displays the following information about each file:

```
```
Displaying Information from Directories 93

- type of file
- permissions
- number of links to the file
- name of userid that owns the file
- name of the group with which the file is associated
- size of the file (in bytes)
- date that the file was last modified
- name of file

Here are some examples, selected from different directories:

```
-rw-r--r--  2  harley  group   9940  May 17 23:44 /usr/harley/myfile
-drwxr-xr-x  1  dianne expert     48  May  2 10:11 /usr/dianne
-c-w--w--w-  1  bin    bin      6,1  Jun 11 13:49 /dev/lp1
-brw-rw-rw-  3  bin    bin      2,11 May 26 13:44 /dev/fd1
```

The `-l` option displays eight columns of information about each file. The first column has ten characters. The first character tells what kind of file is being described:

```
- ordinary file
  d directory
  c or b special file
```

There are two types of special files. The ones marked `c` process data one character at a time. The ones marked `b` process data in blocks. You can ignore the distinction. The next nine characters show the permissions for the file. (Permissions are discussed in Chapter 17.)

The second column shows the number of links to the file. As I explained in Chapter 7, each link refers to an entry in a directory. Notice that the first file has 2 links.

The third column shows the name of the userid that owns the file. The fourth column shows the name of the group associated with the file. (Groups are discussed in Chapter 17, with permissions.)

The fifth column shows the size of the file in bytes. In text files, each byte holds one character. (For special files, you will see two numbers that you can ignore. They are of interest only to advanced programmers.)

The last three columns show the date and time that the file was last modified and the name of the file.
The `lc` Command—More Options

This section describes a few more options for the `lc` command. If you are reading this chapter for the first time, you might want to skip this section until you are more experienced.

The `-F` option flags any files that are directories or that contain programs that can be executed. (F stands for “flag.”) `lc` will display a special character after the name of each such file: `/` (slash) for directories, `*` for programs. Here is an example:

```
lc -F book
```

dispays:

```
archive/  chapter.1  chapter.2  test*
```

In this example, `archive` is a directory, and `test` holds a program that can be executed. When you use this option, be sure to use an upper-case `F`; a lower-case `f` will give you a different option.

Normally, `lc` does not display the names of files that begin with a period. For example:

```
.profile
```

The `-a` option displays the names of all files, including those that start with a period.

You might wonder why you would give a file a name that starts with a period. Well, you do it to take advantage of exactly this feature of the `lc` command. Some files are for special purposes, and you don’t want to see their names every time you check your directory. If you start the names with a period, you don’t see them unless you specifically ask for them with the `-a` option.

The `-R` option processes every sub-directory that it comes across. For example, let’s say that `book` is a directory that contains two sub-directories, and you enter:

```
lc -R book
```

First, `lc` will display the names of the files in `book`. Then `lc` will do the same for the two sub-directories. If either of them contains a sub-directory of its own, the process will go on. The net result is that `lc` will follow every possible outward branch of the tree, starting from `book`.

Here is another example: If you wanted to display the names of all the files in the whole tree, you could use the `-R` option and start from the root directory:

```
lc -R /
```
When you use this option, be sure to type an upper-case \texttt{R}; a lower-case \texttt{r} will give you a different option.

Usually, when you specify the name of a directory, \texttt{lc} will display information about the files in that directory. The \texttt{-d} option displays information about the directory itself. Consider these two examples:

\begin{verbatim}
\texttt{lc -l book}
drwxr-xr-x 2 harley group 32 May 22 23:46 archive
-rw-r--r-- 1 harley group 12133 May 29 21:04 chapter.1
-rw-r--r-- 1 harley group 18794 Jun 7 16:26 chapter.2
-rwxr--r-- 1 harley group 39 May 23 13:21 save
\end{verbatim}

\begin{verbatim}
\texttt{lc -ld book}
drwxr-xr-x 3 harley group 656 Jun 12 13:20 book
\end{verbatim}

The first example displays information about the files in the directory \texttt{book}. The second example displays information about \texttt{book} itself.

\textbf{The \texttt{ls} Command}

As I mentioned earlier, the \texttt{ls} command is an old form of the \texttt{lc} command. Although the commands are similar, they are just different enough to mix you up if you use both of them. The best thing is to use \texttt{lc} all the time except for one particular case.

The \texttt{lc} command displays file names in columns, while the \texttt{ls} command displays file names on separate lines. Occasionally, you may want the file names on separate lines. In this case, you can either use \texttt{lc} with the \texttt{-l} option (the number \texttt{1}) or use \texttt{ls}. For example, the following commands are equivalent:

\begin{verbatim}
\texttt{lc -l book}
\texttt{ls book}
\end{verbatim}

If you don't specify a name, the \texttt{ls} command uses your working directory, just like the \texttt{lc} command. Thus, these two commands are equivalent:

\begin{verbatim}
\texttt{lc -l}
\texttt{ls}
\end{verbatim}

So here is the only advantage of the \texttt{ls} command: When you want to display file names on separate lines, it is slightly faster to type "\texttt{ls}" than to type "\texttt{lc -l}".

For completeness, here is the syntax for the \texttt{ls} command, showing the important options:
Is [-adl] [name...]  
The options have the same meaning as with the lc command. With the -l option (the letter l), both commands display the same "long" form of output.

Special Names for a Directory and Its Parent Directory

Whenever you make a new directory, Xenix puts in two special names. The first name consists only of a period; the second name consists of two periods. You will see these names when you use the lc command with the -a option (see above). For example:

<table>
<thead>
<tr>
<th>Options</th>
<th>Permissions</th>
<th>User</th>
<th>Group</th>
<th>Size</th>
<th>Date</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>drwxr-xr-x</td>
<td>3</td>
<td>harley</td>
<td>group</td>
<td>656</td>
<td>Apr 12</td>
<td>book</td>
</tr>
<tr>
<td>drwxr-xr-x</td>
<td>6</td>
<td>harley</td>
<td>group</td>
<td>368</td>
<td>Apr 12</td>
<td>.</td>
</tr>
<tr>
<td>drwxr-xr-x</td>
<td>2</td>
<td>harley</td>
<td>group</td>
<td>32</td>
<td>May 22</td>
<td>archive</td>
</tr>
<tr>
<td>-rw-r--r--</td>
<td>1</td>
<td>harley</td>
<td>group</td>
<td>12133</td>
<td>May 29</td>
<td>chapter.1</td>
</tr>
<tr>
<td>-rw-r--r--</td>
<td>1</td>
<td>harley</td>
<td>group</td>
<td>18794</td>
<td>Jun 7</td>
<td>chapter.2</td>
</tr>
<tr>
<td>-rwxr--r--</td>
<td>1</td>
<td>harley</td>
<td>group</td>
<td>39</td>
<td>May 23</td>
<td>save</td>
</tr>
</tbody>
</table>

Notice the names of the first two files; they are usually pronounced "dot" and "dot-dot," respectively. To avoid confusion, I will use these names in this book.

"Dot" represents the directory itself; "dot-dot" represents the parent of the directory. For example, if the full name of the directory is /usr/harley/book these two files represent:

. /usr/harley/book
.. /usr/harley

At all times, you can use "dot" to refer to the working directory itself. For example, if the working directory is /usr/harley/book, the following three commands are equivalent:

lc /usr/harley/book
lc .
lc

(Remember, if you don't specify a file name, the lc command will use the working directory.)
You can use "dot-dot" to refer to the parent directory. With the same working directory, consider these two equivalent commands:

lc /usr/harley
lc ..
Think of “dot-dot” as an easy way to move one step toward the root of the tree. In the next example, both commands specify the same file. The second example uses the parent of the parent, starting from /usr/harley/book:

```
lc /usr
lc ../..
```

You can even back up all the way to the root directory. For example, starting from /usr/harley/book, these two commands are equivalent:

```
lc /
lc ../.. ../..
```

The “dot-dot” file is especially handy when you want to reference a file in the parent directory without having to change your working directory. For example, say that /usr/harley has two sub-directories: book and work. The working directory is currently /usr/harley/book and you want to display the names of the files in /usr/harley/work. One way would be as follows:

```
lc /usr/harley/work
```

But an easier way is:

```
lc ../work
```

Here is one last example: The working directory is still /usr/harley/book. Here are two equivalent commands that refer to the file /usr/dianne/letter:

```
lc /usr/dianne/letter
lc ../../dianne/letter
```

After a bit of practice, you will find it easy to use commands like the second one.

**PRACTICING THE DIRECTORY COMMANDS**

**A Sample Session Using the Directory Commands**

This section contains examples that illustrate the **pwd, cd, lc, and ls** commands. Before you continue reading, log in so you can follow along at your terminal.

These commands are among the most important in all of Xenix. Before you enter the example, think about what it is supposed to do. By
the time you finish this session, you will have acquired a good amount of facility in using the directory commands.

When you log in, the shell will automatically set your working directory to be your home directory. To see what it is, enter the `pwd` command:

```
pwd
```

Unless your system manager has changed things, your home directory will be a sub-directory of `/usr`, with the same name as your userid. For example, if your userid is `dianne`, your home directory will be `/usr/dianne`.

You will now use the directory commands to explore the file system. Change your working directory to the root directory and use the `pwd` command to make sure it worked properly.

```
cd /
pwd
```

Display the files in the root directory.

```
lc /
```

Enter the `lc` command without a name. `lc` will use your working directory. Because this is the root directory, the results will be the same.

```
lc
```

Display the files again, but this time use the `-F` option to flag the directories and executable files. (Be sure to type an upper-case `F`; a lower-case `f` will give you a different option.)

```
lc -F
```

Remember that the `-F` option displays a `/` after the name of each directory, and a `*` after the name of each executable file. Notice that most of the files are directories. Change to the `bin` directory and use `pwd` to check. Enter both commands on one line.

```
cd bin ; pwd
```

Display the names of the files in this directory using the `-F` option.

```
lc -F
```

Notice how many files are flagged with a `*`. This directory contains many of the programs that carry out the Xenix commands. Since these programs are stored as binary files, the directory that contains them is called `/bin`. There is another directory called `/usr/bin` that contains more such files.
You will now change back to the root directory. One way would be to enter `cd /`. However, remember that you can refer to the parent directory as "dot-dot." Since you are now in `/bin`, the parent directory is the root directory. Change to it using "dot-dot."

```
    cd..
```

Check to make sure it worked.

```
pwd
```

Display the file names again with the `-F` option.

```
lc -F
```

Notice the `etc` directory. Change to it, and see what it contains. Use the `-F` option again.

```
    cd etc ; lc -F
```

As you see, the `etc` directory contains a sub-directory called `default`. Change to this directory and display the names of its files.

```
    cd default ; lc
```

Display the file names again, but this time, use the `-a` option to display "all" the file names.

```
    lc -a
```

Notice that the first two names are "dot" and "dot-dot." Now, display each file name on a separate line. (Note: the option is the number 1.)

```
    lc -l
```

Do the same using the `ls` command.

```
    ls
```

Notice that the output is identical.

Use the `lc` command with the `-l` option to display more information about each file (the "long" display). (Note: the option is the letter l.)

```
    lc -l
```

Use `pwd` to remind yourself where you are. Then, change to the parent directory and check again.

```
pwd
    cd .. ; pwd
```

You should be in `/etc`. You will now use `lc` with the `-d` option. Enter the following two commands.
Using the Xenix File System

```
lc -d
lc
```

Notice that the first command displays information about the working directory itself (/etc), while the second displays information about the files in the working directory.

Now, use the `lc` command with the `-R` option to examine the working directory and its sub-directories. (Be sure to type an upper-case `R`; a lower-case `r` will give you a different option.)

```
lc -R
```

Notice that the sub-directory `default` was automatically processed.

Enter a similar command starting from the root directory. This command will display the files in every directory in the system. The output will be displayed so fast that you will have trouble reading it. Remember that you can press `<CTRL-s>` to pause the display and `<CTRL-q>` to continue. If you want to stop the command before it finishes, press `<DEL>`. Now,

```
enter: lc -R /
```

Use `pwd` to remind you where you are; you should be in `/etc`.

```
pwd
```

Practice jumping from directory to directory. After each `cd` command, use `pwd` to check where you are.

```
  cd default
  pwd
  cd /bin
  pwd
  cd /etc/default
  pwd
  cd ..
  pwd
```

Now, end the session by changing to your home directory. You can do this by entering the `cd` command without specifying a name.

```
  cd
  pwd
```

Remember this form of the `cd` command. When you are in some far-off branch of the tree, it is an easy way to return to the home directory. (Alternatively, close your eyes, tap your heels together, and repeat to yourself, "There's no place like home..."
CHAPTER 9

USING FILES AND DIRECTORIES

INTRODUCTION

MAKING COPIES OF FILES

MANIPULATING FILES

MAKING AND REMOVING DIRECTORIES

OTHER USEFUL FILE COMMANDS

SPECIFYING A PATTERN OF FILENAMES
INTRODUCTION

In Chapter 8, you learned how to display files and work with directories. In this chapter, you will learn the rest of the important commands for manipulating files. By the time you finish this chapter, you will have a good understanding of the Xenix file system and how to use it.

New Words in This Chapter

REGULAR EXPRESSION
a compact way of specifying all the character strings that match a particular pattern

MAKING COPIES OF FILES

The cp Command

There are two commands to make copies of files: cp and copy. copy is a newer version of cp. Both commands make duplicates of files. The original files remain unchanged.

There are two forms of the cp command. The syntax for the first form is:

```
    cp file1 file2
```

This form of the cp command makes a copy of a file and gives it a different name. The first argument is the name of the original file. It can be either an ordinary file or a special file. The second argument is the name of the destination file. For example, to make a copy of the file file1 and call it file2,

```
    cp file1 file2
```

You cannot copy a file to itself. In other words, you must specify two different names.

The syntax for the second form of the cp command is:

```
    cp file... directory
```

This form of the cp command copies one or more files to a directory. The
new files have the same name. For example, to copy the files file1, file2, and file3 to the directory dir,

```
  cp file1 file2 file3 dir
```

Once the copying is completed, dir/file1 will be a copy of file1, dir/file2 will be a copy of file2, and dir/file3 will be a copy of file3.

Here are two more examples. Both commands copy the file file1 to the directory dir. However, the first command does not change the file name, while the second command does.

```
  cp file1 dir
  cp file1 dir/extra
```

Since the first command copies a file to a directory, the file name remains the same. The new file is dir/file1. The second command names the new file dir/extra.

When you use Xenix, you never have to create a file explicitly. If a destination file does not exist, cp will create it. If a destination file already exists, cp will replace it. During this operation, cp first erases the destination file and then makes the new copy. This means that even if the copying procedure doesn't work, the destination file is still erased.

Here is an example: Say you use the cp command to copy the file trash to the file important.

```
  cp trash important
```

If important does not already exist, cp will create it. If important does exist, the first thing cp will do is erase it. cp will then make a copy of trash and call it important. All the data that was originally in important is gone forever. There is absolutely no way to get it back!

You must be careful: If you enter a cp command and then realize that you are copying to an important file, it is too late. No matter how fast you press <DEL> (to stop the command), cp will have already erased the destination file.

The copy Command

The copy command is a newer version of the cp command. The copy command can do everything the cp command can do and more. The advantages of the copy command are that it can copy a whole directory of files and that it has some useful options.

The copy command has three forms. The syntax is:
copy [-anrv] file1 file2
copy [-anrv] file... directory
copy [-anrv] directory1 directory2

file1, file2, and file must be either ordinary or special files. directory1, directory2, and directory are directories.

Except for the options, the first two forms of the copy command are exactly the same as the cp command. For example, if file1 and file2 are ordinary or special files, and dir is a directory:

    copy file1 file2

is the same as:

    cp file1 file2

and,

    copy file1 file2 file3 dir

is the same as:

    cp file1 file2 file3 dir

Like cp, copy will create destination files if they do not exist and replace them if they do.

You can use the third form of the copy command to copy all the files in a directory to another directory. To do this, specify two directory names as arguments. For example, to copy all the files in the directory dir1 to directory dir2,

    enter:  copy dir1 dir2

The question is: When do you use copy and when do you use cp? Since copy does everything cp does and more, you can use copy all the time. The only advantage to cp is that if you don't need to copy a directory or use an option, it is faster to type "cp" than to type "copy."

The copy Command—Options

The copy command has several useful options:

The -v option displays messages as copy does its work, to let you know what is happening. (v stands for "verbose.")

The -a option will "ask" your permission before it copies a file. For example, say that trash and important are two files. If you enter:

    copy -a trash important
you will see:

```
    copy file trash?
```

If you enter anything that begins with a y (for "yes"), copy will proceed with the copy. If you enter anything else, copy will not perform the copy.

This option is useful for selectively copying particular files out of a group. For example, say that dir1 and dir2 are directories. If you enter:

```
    copy -a dir1 dir2
```

copy will process the files in dir1, one by one. For each file, copy will ask permission to perform the copy. If you want to copy the file, enter y; if not, enter anything else (say, n for "no").

The -n option prevents you from accidentally erasing a file. With this option, copy will not proceed if the destination file already exists. (n stands for "new.") For example, if you want to copy trash to important, but only if important does not already exist,

```
    enter: copy -n trash important
```

The -r option is useful when you are copying directories. This option copies not only each directory but also every sub-directory that it comes across.

For example, say that a whole system of files and sub-directories is stored under /usr/harley. You can copy all of it to another part of the tree. If you wanted to copy it all to /usr/dianne, you would enter:

```
    copy -r /usr/harley /usr/dianne
```

The copy command -r option is analogous to the lc command -R option. Notice, however, that the lc option is an upper-case R.

**MANIPULATING FILES**

The ln Command

In Chapter 7, I explained that directories do not really contain files, they contain links to files. Xenix stores the actual files elsewhere. A link
consists of a name and a file identification number, called an inumber. Xenix automatically handles the details.

Occasionally, it is useful to have the same file appear in different directories. In other words, to have more than one directory contain a link to the same file.

The `ln` command creates a link to an existing file. The syntax is:

```
ln  file1  file2
```

The first argument is the name of the existing file. The second argument is the name of the new link.

Here is an example: Say that you want to create a link to the file `/usr/harley/test`. The link is to be called `/usr/dlanne/mytest`. Use the command:

```
ln  /usr/harley/test  /usr/dlanne/mytest
```

If the working directory is `/usr/dlanne`, then the following commands are equivalent to the first one:

```
ln  /usr/harley/test  mytest
ln  ./harley/test  mytest
```

The result of the command is that the two directories, `/usr/harley` and `/usr/dlanne`, both contain links to the same file. The first directory knows the file as `test`, while the second directory knows the file as `mytest`.

The important thing to realize is that although there are two names, there is only one file. If one user changes the file under one name, another user will notice the changes even if he or she accesses the file under a different name.

Once the links are established, Xenix does not distinguish between the original directory entry and subsequent links. However, the owner of the file does not change; and the owner is the one who retains full control over the file. (This is explained in Chapter 17, with permissions.)

You can find out how many links a file has by using the `lc` command with the `-l` option (the letter `l`).

Linking is useful if more than one user needs to access the same file. Here is an example: A company has a very large file containing information about all their customers. Ten different users need to access the file for their work. However, the file is so large that it would take up too much disk space to keep ten copies. As well, it would be inconvenient to have to change ten copies every time the file is updated.

The solution is to use links: One user creates and maintains the file under his or her userid and is considered the owner. All the other users
link to the file from their own directories, under any file names they want. If necessary, the owner can set the permissions so that the other users can read from, but not write to, the file.

In order to maintain the integrity of the file system, Xenix prohibits you from making two types of links. First, you cannot link to directories because this would play havoc with the structure of the tree. (Take a moment and think about it.) Thus, you can link only to ordinary or special files.

Second, you cannot link across file systems. File systems are explained in Chapter 15. For now, it suffices to know that from within the /usr part of the tree, you cannot link to files outside of /usr, nor can you link to files on a diskette or tape.

The **rm** Command

The **rm** command removes a link to a file. Use this command to erase a file. The syntax is:

```
rm [-ir] file . . .
```

Here is an example. This command removes the link, in the working directory, to the file **temp**.

```
rm temp
```

Most files only have one link. In such cases, the **rm** command removes the only link to the file. In other words, the **rm** command erases the file. *Once a file is erased, it is gone forever, and there is no way to get it back.* Thus, you must be most careful of how you use the **rm** command.

Here is an example: Say that the file **data**, in the directory /usr/harley, has only one link. If /usr/harley is the working directory, either one of the following commands will erase the file:

```
rm /usr/harley/data
rm data
```

If a file has more than one link, the **rm** command removes only one of those links. The file will still exist—accessible under a different name—until the last link is removed.

If you use the **-i** option, **rm** will ask you for permission before it removes each file. (**I** stands for “interactive.”) If you respond with anything that begins with **y** (for “yes”), **rm** will remove the file. If you respond with anything else (say, **n** for “no”), **rm** will not remove the file.

Normally, **rm** will remove only ordinary or special files. If you want
rm to remove a directory, use the -r option. rm will remove all the files in the directory, along with the directory itself.

Obviously, this is a very powerful option. There is another command, rmdlr (see below) that removes directories. rmdlr is the better one to use because it has built-in safeguards to keep you from accidentally causing a catastrophe.

In order to maintain the integrity of the tree structure, rm will not remove a “dot-dot” (..) file or the root directory of a file system. (File systems are explained in Chapter 15.)

The mv Command

The mv command moves the link that points to a file. Use this command to rename or move a file. The mv command has three forms. The syntax is:

```
mv file1 file2
mv directory1 directory2
cmp file... directory
```

file1, file2, and file are ordinary or special files. directory1, directory2, and directory are directories.

You can use the first form of the mv command either to rename a file or to move a file to a new directory. If you specify two file names in the same directory, mv moves the link from the first name to the second. In other words, mv renames the file.

For example, if the working directory is /usr/harley, either of the following commands will rename /use/harley/temp to /usr/harley/keep:

```
mv /usr/harley/temp /usr/harley/keep
mv temp keep
```

If you specify two files in different directories, mv moves the first file to the second. For example, if you want to move the file /usr/harley/myfile to /usr/dlanne/yourfile, use the command:

```
mv /usr/harley/myfile /usr/dlanne/yourfile
```

Like the copy and cp commands, mv first checks if the destination file already exists; if so, mv erases it before moving the link. For example, say that the files trash and important both exist, and you enter:

```
 mv trash important
```

You will end up with one file, named important, containing whatever
was in \texttt{trash}. The information originally in \texttt{important} will be lost.

Also like the \texttt{copy} and \texttt{cp} commands, you must specify two different file names. In other words, you cannot move a file to itself.

You can rename a directory by using the second form of the \texttt{mv} command. For example, say you want to rename the directory \texttt{/usr/harley/save} to \texttt{/usr/harley/archive}. If \texttt{/usr/harley} is the working directory, either of the following two commands would do the job.

\begin{verbatim}
mv /usr/harley/save /usr/harley/archive
mv save archive
\end{verbatim}

\texttt{mv} renames the directory by moving the link in the parent directory to a different name. The contents of the files in the directory are not changed.

The third form of the \texttt{mv} command moves the links for one or more files to a new directory. For example, if you want to move the files \texttt{file1}, \texttt{file2}, and \texttt{file3} from the working directory to the directory \texttt{/usr/dianne/extra}, use the command:

\begin{verbatim}
mv file1 file2 file3 /usr/dianne/extra
\end{verbatim}

When you use the \texttt{mv} command, it is up to you to know which files are directories and which are ordinary or special files. If you are not sure, examine the files using the \texttt{lc} command.

**MAKING AND REMOVING DIRECTORIES**

The \texttt{mkdir} Command

The \texttt{mkdir} command makes a new directory. The syntax is:

\begin{verbatim}
mkdir directory...
\end{verbatim}

The argument is the name of the new directory. For example, the following command creates the directory \texttt{/usr/harley/archive}.

\begin{verbatim}
mkdir /usr/harley/archive
\end{verbatim}

If the working directory is \texttt{/usr/harley}, you could use:

\begin{verbatim}
mkdir archive
\end{verbatim}

\texttt{mkdir} automatically puts the "dot" and "dot-dot" (self and parent) entries in every new directory. You can display information about these entries by using the \texttt{lc} command with the -\texttt{a} option.
You cannot create a new directory if its parent directory does not exist. For example, let us say that you have not yet created the directory `/usr/harley/book`. `mkdir` will not create `/usr/harley/book/chapters`. If you did enter:

```
mkdir /usr/harley/book/chapters
```

`mkdir` would display the message:

```
cannot access /usr/harley/book
```

The idea is to plan your file system so that the directory structure matches the underlying organization of your files. You then use `mkdir` to create the necessary directories. For an example of how to plan such a system, see the section “Organizing with Sub-Directories” in Chapter 7.

**The rmdir Command**

The `rmdir` command removes a directory. The syntax is:

```
rmdir directory...
```

This command is straightforward. For example, say that the working directory is `/usr/harley/book`, and you want to remove the directory `/usr/harley/book/archive`. You can use either of the following commands:

```
rmdir /usr/harley/book/archive
rmdir archive
```

In order to protect you from mistakes, `rmdir` has two rules: You may not remove your working directory, and you may not remove a directory that is not empty. The first rule maintains the integrity of the tree. If you could remove your working directory, it would be like sitting in a tree and sawing off the branch on which you are sitting. The second rule guards against accidental catastrophe, by forcing you to empty a directory before you remove it.

For example, say that you have two directories, `frick` and `frack`. `frick` has many files and sub-directories, while `frack` is empty. Since `frack` is empty, you want to remove it, but by accident you enter:

```
rmlr frick
```

`rmdir` will display:

```
frick not empty
```

If `rmdir` had removed the nonempty `frick`, you would have lost all of your files and sub-directories.
With this in mind, let us compare the `rmdlr` command with the `rm` command using the `-r` option. The `rmdlr` command will remove only empty directories. The `rm -r` command will remove any directory, including all of its files and sub-directories. Clearly, you should use `rmdlr` most of the time. Here is an example to convince you.

Say that your home directory is `/usr/cimarron`. You have an empty directory, `/usr/cimarron/accounts/current`, that you want to remove. You should use `rmdlr`, but instead you use `rm -r`. “I know that the directory is empty,” you reason, “so what difference could it make? Besides, it takes so long to type 'rmdlr'.”

But, when you enter the command, you make a typing mistake and insert a space after “cimarron.”

```
rm -r /usr/cimarron/accounts/currents
```

It looks to `rm` as if you want to remove two directories. The result is that you have just erased `/usr/cimarron`, including all of its files and sub-directories. Imagine what destruction the superuser could cause by entering:

```
rm -r /usr/cimarron/accounts/current
```

Here's the rub: As a habit, use `rmdlr` to remove directories. If you are completely sure that you want to remove a directory and all its files and sub-directories, use `rm -r` because it is a lot faster—but be careful.

OTHER USEFUL FILE COMMANDS

The `find` Command

The `find` command searches through directories and looks for a file with a particular name. This command is useful when you are trying to find a file, but you can't remember where it is.

`find` is a complex command with a syntax that is different from other commands. In this section, I will describe only the most basic form of the command. For more details, see the Xenix Command Reference manual.

The syntax of the basic form of the `find` command is:

```
find directory -name file -print
```

where `directory` is the name of the directory from which the search is to
Using Flies and Directories

start. **find** searches this directory, as well as all of its sub-directories. Next, comes the string **-name** followed by the name of the file you want to find. Last comes the string **-print**. This tells **find** to display the names of any files it finds. (Remember, the Xenix tradition is to use the word "print" to mean "display.") If you don't specify **-print**, **find** will not display anything.

Here is an example. To find the pathnames of all the files named **test**, starting from the directory `/usr/harley`,

```
find /usr/harley -name test -print
```

**find** will display the names of the files it finds. For example:

```
/usr/harley/book/test
/usr/harley/book/chapter.1/test
/usr/harley/test
/usr/harley/archive/test
```

If no files are found, **find** will not display anything. However, if this happens, make sure that you have not inadvertently left out the **-print** part of the command.

Remember, you can use "dot" to represent your working directory. For example, if your working directory is `/usr/harley`, the following two commands are equivalent:

```
find /usr/harley -name test -print
find . -name test -print
```

If you use "dot" to specify the directory, **find** will use "dot" in the pathnames it displays. For example:

```
./book/test
./book/chapter.1/test
./test
./archive/test
```

Here is an example that you can try for yourself:

```
find / -name bin -print
```

You will see:

```
find: cannot open /usr/spool/at
find: cannot open /usr/spool/uucp/.xqtdlr
find: cannot open /usr/spool/uucp/.xfddlr
/bin
/usr/bin
```
You can ignore the first three lines. They are error messages because find tried to examine directories that you do not have permission to search. (Permissions are explained in Chapter 17.) The last two lines are the pathnames of the files named bin. (These two files happen to be the directories that contain most of the programs that interpret the Xenix commands.)

The **file** Command

The **file** command tells you what sort of information is in a file. The syntax is:

```
file file...
```

You specify the names of one or more files. **file** examines the files and classifies them. For example, to classify the file **letter**,

```
enter: file letter
```

Here is an example followed by a sample of what **file** might display. This example specifies four file names.

```
file /usr letter prog.c temp
    /usr: directory
    letter: English text
    prog.c: c program text
    temp: ascii text
```

The first file is a directory and the other three files are text files. **letter** is English and **prog.c** is a program written in the computer language C. The last classification, **ascii text**, is what **file** calls text files when it can't figure out their type.

Note: **file** does its best, but it is sometimes wrong. If you want more information about a file, use the **lc** command with the -l option (the letter l).

The **wc** Command

The **wc** command counts the number of lines, words, and characters in a file. The syntax is:

```
wc [-clw] [file...]
```
This command is straightforward. You specify one or more files, and `wc` displays the number of lines, words, and characters in the file, followed by the name of the file. For example, if `letter` is a file, and you enter:

```
wc letter
```

`wc` might display:

```
283  2215  12317  letter
```

In this case, `letter` has 283 lines, 2,215 words, and 12,317 characters.

`wc` assumes that each line is terminated by a `newline` character and that each word is terminated by a `space`, `tab`, or `newline` character.

If you are not interested in all three numbers, you can specify the ones you want by using the options: `-c` for characters, `-l` for lines, and `-w` for words. Here are two examples:

```
w -w letter
wc -cl letter
```

The first command counts only words, and the second command counts only characters and lines.

**SPECIFYING A PATTERN OF FILE NAMES**

Regular Expressions

A REGULAR EXPRESSION is a compact way of specifying all the character strings that match a particular pattern. The term regular expression comes from a branch of theoretical computer science. Regular expressions are one of the fundamental concepts of Xenix. In this section, I will explain how to form them; in the next section, I will explain how to use them to specify a pattern of file names.

A regular expression specifies a pattern that is used to match character strings. For example, you might create a pattern to match strings that consist of the letters `th`, followed by any two characters. Here are some strings that this pattern would match:

```
they  that  them  thxx  thXX  th12  th%%  thG7
```

Within Xenix, there are several different ways of specifying regular expressions. The shell has one set of rules, while other programs have their own sets of rules. (In particular, the `vi` editor—see Chapter 10—
recognizes more complex patterns.) In this section, I will describe how you specify regular expressions for the shell. The other sets of rules are discussed with the other programs.

A regular expression consists of a string of characters. In the string, each character stands for something. Letters and numbers stand for themselves while ?, *, [ ], -, and ! have special meanings.

? stands for any character. For example, the regular expression that specifies th followed by any two characters is:

\( \text{th??} \)

The regular expression that specifies the followed by any three characters is:

\( \text{the??} \)

The regular expression that specifies any character, followed by x, followed by any two characters, followed by yz, followed by any three characters, followed by 2 is:

\( ?x??yz???2 \)

Here are some examples of strings that this last regular expression would match:

- AxAAyzAAA2
- mxopyz1232
- qxyzyz2222

* stands for zero or more occurrences of any character. In other words, * will match nothing, or any number of characters. Here is an example: The following regular expression specifies th followed by zero or more characters:

\( \text{th*} \)

Here are some strings that this pattern would match:

- they
- that
- them
- thXXX
- th12345
- thabcdefghijklmnop

Here is the regular expression to specify th, followed by zero or more characters, followed by e:

\( \text{th*e} \)

Here are some strings that this pattern would match:

- the
- thXe
- thXXXXXe
- th1234567890e

The * is more powerful than you might realize at first. For example, * by itself will match any string; s* will match any string that begins with s; ss will match any string that ends with s.
You can use both * and ? in the same regular expression. For example, to specify any string that starts with s, followed by any two characters, followed by t, followed by zero or more characters, you would see:

```
s??t*
```

Here are some strings that this pattern would match:

```
sXXtXXXX sXXt sabtefg setting seat s111234567890
```

Here is a more complicated regular expression that specifies any string that starts with zero or more characters, followed by x, followed by any character, followed by bb, followed by any character, followed by cc, followed by zero or more characters:

```
*a?bb?cc*
```

Here are some strings that this pattern would match:

```
XaXbbXccX aXbbXcc 12345aqbbwcc12345
```

* by itself matches zero or more characters. If you want to match one or more characters, use ? followed by *· Here is a regular expression that specifies t followed by one or more characters:

```
t??
```

The ? matches one character, and the * matches zero or more characters. Together, they match one or more characters.

Sometimes you will want to specify that a character must belong to a particular set of characters. For example, you may want to match strings that start with chapter followed by either 1, 2, or 3. You can do this by enclosing your choices in [ and ] like this:

```
chapter[123]
```

This regular expression matches:

```
chapter1  chapter2  chapter3
```

You can use this construction anywhere in a regular expression. The whole thing, [, the characters inside, and ], represents only one character. Here is another example: To match the strings that start with a or b, followed by xx, followed by y or z, use:

```
[ab]xx[yz]
```

This regular expression matches:

```
axxy  azz  bxy  bxxz
```
As an abbreviation, you can use the character - to represent a range of consecutive single numbers or letters. For example, the following two regular expressions are equivalent. Both specify the numbers 0 through 9:

- [0123456789]
- [0-9]

Remember, - only defines a range of single numbers or letters. Consider the following regular expression:

- [1-36]

This does not match the numbers 1 through 36; it matches 1 through 3 or 6. In other words, 1 or 2 or 3 or 6.

You can also use - with letters. For example, both of the following regular expressions match the letters a through f:

- [abcdef]
- [a-f]

The next example of a regular expression specifies th, followed by a through f, followed by xx:

- th[a-f]xx

This regular expression would match:

thaxx thbxx thcxx thdxx thexx thfxx

You can use * after a set within [ and ] to indicate zero or more members of the set. For example, to match any string that starts with chapter, followed by zero or more of the numbers 0 through 9, use:

- chapter[0-9]*

Here are some strings that this regular expression would match:

- chapter chapter0 chapter1 chapter11 chapter1234512345

This regular expression matches chapter followed by any upper-case letter:

- chapter[A-Z]

The next example matches chapter followed by zero or more upper-case letters:

- chapter[A-Z]*

For example, it would match:
chapter chapterA chapterBC chapterDEF

Notice that chapter is matched, because * means zero or more. If we want to match one or more upper-case letters, we would use:

chapter[A-Z][A-Z]*

The first [A-Z] matches exactly one upper-case letter, and [A-Z]* matches zero or more upper-case letters. So this regular expression would match chapterA, chapterBC, and chapterDEF, but not chapter.

The last character that has a special meaning is !. You can use ! directly after a [ character, when you are specifying a set of characters. This construction matches any character that is not in the set. For example, the regular expression:

xx[lab]yy

matches the characters xx, followed by any character that is not a or b, followed by yy. Thus, it would match:

xxzyy zz1yy zzAyy

but not:

xxayy xxbyy

Here is a summary of the rules for specifying regular expressions for the shell:

* matches zero or more characters
? matches any one character
[ ] matches one of the enclosed characters
[!] matches any character that is not enclosed

Inside [ and ], you can use – to indicate a range of single letters or digits.

When Do You Use Regular Expressions?

A regular expression is a short way of specifying strings that match particular patterns. You will find that Xenix allows you to use regular expressions anywhere it makes sense. For example, when you use an editor to create and modify a text file, you can use regular expressions to specify strings that you want to change or delete.

One of the most important uses of regular expressions is with the file commands that you learned in the previous chapters. Xenix allows you to use a regular expression to specify a set of file names, whenever it makes sense to do so.
For example: If you wanted to display information about the files `chapter1`, `chapter2`, `chapter3`, `chapter4`, `chapter5`, and `chapter6`, you could use:

```
lc -l chapter1 chapter2 chapter3 chapter4 chapter5 chapter6
```

However, it is a lot easier to match the pattern using a regular expression:

```
lc -l chapter[1-6]
```

When you use a regular expression in a Xenix command, the shell replaces it with all the files in the appropriate directory that match the pattern. (In this case, the working directory.) Here is an example of a command that copies all the files whose names begin with `s` from the directory `/usr/harley/book/` to the directory `/usr/dlanne`:

```
cp /usr/harley/book/s* /usr/dlanne
```

If the working directory is `/usr/harley/book`, the following two commands have the same effect:

```
cp s* /usr/dlanne
    cp s* ../..../dianne
```

`*` by itself will match all the files in the directory. Thus, to copy all the files from the working directory to `/usr/dlanne`, use:

```
cp * /usr/dlanne
```

Be very careful when you use regular expressions with destructive commands. For example:

```
rm *
```

will remove all the files in your working directory. Remember, once a file is erased, you can't get it back.

**Quoting with a Backslash**

There will be times when you want to use a special character within a regular expression. For example, how would you specify the string starting with `acb?` followed by any character? If you used:

```
abc??
```

it would match `abc` followed by any two characters. The solution is to enclose the first `?` in single-quotes:
This tells the shell that ? is to be taken literally; not as a pattern-matching character. One problem with using single-quotes is that they become confusing if you use them more than once. Look at the following regular expression that matches ab?, followed by any character, followed by ?, followed by any character, followed by ?ab:

ab'??'?'?'?'ab

And what if you want to match ab'ab? It would be confusing to use single-quotes around a single-quote.

To overcome this problem, the shell lets you quote a single character by preceding it with a backslash (\). This has the same effect as enclosing the character in single-quotes. For example, here is a regular expression that is equivalent to the last one:

ab\??\??\??ab

It is a lot easier to understand; just remember that each character after a backslash is to be taken literally. Occasionally, you will have an expression so complicated that it is confusing even when you use backslashes instead of single-quotes. There is not much you can do about it except to type slowly and carefully.

Using a backslash, you quote any special character, including another backslash. For example, the following two regular expressions match any string starting with ab, followed by a backslash, followed by ab, followed by any character:

ab'\ab?
ab\ab?

The first backslash means that the second one is to be taken literally. When necessary, you can quote a single-quote by preceding it with a backslash.

As with single-quotes, you can use the backslash any time you are entering information to the shell and not only within regular expressions. For example, you know that you can enter more than one command at a time by separating them with semicolons. For instance:

cd /usr/dlanne ; lc

What if you wanted to use echo to display Hello; goodbye? If you entered:

echo Hello; goodbye

the shell would treat it as two commands. The first command would
display **Hello**, and the second command will not be found. To display the semicolon, you must quote it. Here are three commands that will do so:

```bash
echo 'Hello; goodbye'
echo Hello';' goodbye
echo Hello\; goodbye
```

It is your preference as to which one you use. However, it is a good idea to prefer backslashes unless you are quoting more than one character. Compare these two equivalent commands. Clearly, it is easier to use single-quotes in this case:

```bash
echo The characters `?<` are important
echo The characters `\?<` are important
```

In this example, you need to quote `?<` because they have a special meaning to the shell.

Whenever you enter a command that uses special characters literally, and you are not sure if those characters have a special meaning, play it safe and quote them, either with single-quotes or backslashes.
CHAPTER 10

THE vi EDITOR—THE BASICS

INTRODUCTION

STARTING TO USE vi

WORKING WITH vi

MANIPULATING YOUR WORK FILES
INTRODUCTION

By far the most important part of Xenix that you must learn how to use is the editor. This is the program that allows you to create and modify your own text files. The editor that comes with Xenix is powerful and complicated. This chapter introduces you to the editor and covers the most important aspects of getting started. The next three chapters cover the day-to-day details.

New Words in This Chapter

SCREEN EDITOR
an editor that makes use of the whole screen to allow you to see and work with more than one line at a time

LINE EDITOR
an editor that does not make full use of the screen, but rather works with and displays groups of lines

CURRENT FILE
the text file that you are working with at a particular time

EDITING BUFFER
a working copy of the current file

COMMAND LINE
the bottom lines of the screen, used by \texttt{vi} to display messages and commands

MODE
a particular way in which a program can work

COMMAND MODE
the mode in which you can use editor commands

INSERT MODE
the mode in which you can add information to the editing buffer

The Editors That Come with Xenix

As you have already learned, an editor is a program that you use to create and modify text files. When you are using such a program, we say that you are editing the file.
You might wonder how an editor differs from a word processing program. The answer is that editors are not as powerful. Word processors can perform operations that an editor cannot: automatic pagination, page formatting, spelling checking, and so on. Xenix can perform all the functions of a word processor and more. However, in keeping with the philosophy of one tool for one task, these functions are performed by different programs. The editors confine themselves to creating and modifying files.

There are three editors that come with Xenix: vi, ex, and ed. (The names are pronounced: “vee-eye,” “ee-ex,” and “ee-dee.”)

vi is a SCREEN EDITOR. This is an editor that makes use of the whole screen to allow you to see and work with more than one line at a time. When you use vi with an IBM PC AT computer, you can see and work with 24 lines at a time.

ex and ed are LINE EDITORS. Line editors were developed in the days before display terminals. In those days, terminals were typewriter-like devices that printed one line at a time on paper. (This is why the Unix tradition is to use the term print to refer to displaying information at a terminal.) Line editors were developed to work in this environment—they deal with information in terms of lines, and they do not take full advantage of the screen to display your file. In these days of television-like terminals, screen editors are much more convenient than line editors.

vi and ex are really different faces of the same program. As you use vi, you can enter ex commands whenever you want. ed is an older version of ex. It is the original editor that was used with Unix.

What editor will you use? Because a screen editor is more convenient, you will use vi almost all the time. If you encounter a situation in which an ex command is called for, you can enter the command without leaving vi.

Occasionally, you will want to use a line editor exclusively. For example, you may be using a terminal so slow that it takes too long to continually update a whole screen. In such a case, as with the old terminals, a line editor is more convenient, and you will use ex. ed is obsolete—you will never need to use it.

STARTING TO USE vi

Learning vi

Once you learn vi, you will like it. You will find that it is easy to use but difficult to learn.
Starting to Use vi

There are a great many commands, and at the beginning you may be overwhelmed. As well, vi provides a particular working environment that you will have to get used to. vi is different from any other editor or word processor, so even if you are an experienced computer user, don't get discouraged if you feel lost at first.

You will find that you can handle most tasks in more than one way. Indeed, there are often five or six ways to do something. The art of using vi is to understand all the different commands so well that you can instantly choose the best one to use at any particular time.

Of course, this will take awhile. As with other skills, you must practice. With vi, this means that you must go out of your way to add new commands to your repertoire. It is possible to get by with only a few commands; however, to do so is a mistake. It is not until you have a good many commands at your fingertips that you will appreciate the power and ease of vi. Fortunately, vi commands are simple, so it will not take you a long time to become proficient.

Here is how you should learn vi: Read this chapter slowly, and as you do, try each new command at your terminal. Do not expect to remember all the commands right away. Just make sure you use each command at least once.

After you have finished the chapter, start using vi with only a few commands. I have included a summary of vi as an appendix to this book. As you work, keep this summary open beside your terminal. Start with a few commands from each group and use only those commands until you are comfortable with them. Every few days, force yourself to learn a few new commands. I strongly urge you to continue learning until you are familiar with almost all the commands.

Starting vi

To start vi, use the vi command. There are several options, but I will list only the most useful ones. The syntax is:

```
vi [-rR] [filename...]
```

where filename is the name of the file that you want to edit. You can specify more than one name. vi will start with the first one, and you can switch to one of the others whenever you want. The file that you are editing is called the CURRENT FILE.

If the current file does not exist, vi will create a new file and give it that name. If you do not specify a file name, vi will let you work with an unnamed current file. You can give the file a name when you save it.

Most of the time, you will start vi with no options and with one file name. For example, to edit the file letter,
**enter:**  \texttt{vi letter}

\texttt{vi} does not really work with the current file; rather, \texttt{vi} makes a copy of the file and stores it in a work area called the \textit{EDITING BUFFER}. \texttt{vi} works only with the editing buffer, not with the original file. Before you stop \texttt{vi}, you can save your work. \texttt{vi} will copy the editing buffer to the file. On the other hand, you can choose to not save the work. \texttt{vi} will discard the editing buffer, and your original file is left untouched.

The \texttt{vi} command has two important options. The first, \texttt{-r}, stands for "recover." Occasionally, you may be interrupted in the middle of an editing session before you have had a chance to save your work. For example, someone may turn off the power to the computer. When this happens, there is a good chance that \texttt{vi} has been able to save part or all of your work. Enter the \texttt{vi} command with the \texttt{-r} option and no file name:

\begin{verbatim}
  vi -r
\end{verbatim}

\texttt{vi} will display a list of all the files that were saved. If your file was saved, you can recover it by starting \texttt{vi} with the \texttt{-r} option and specifying the name of the file. For example, if you are working with a file called \texttt{letter} and someone turns off the power to the computer, you can recover your work by entering:

\begin{verbatim}
  vi -r letter
\end{verbatim}

The \texttt{-r} option will only recover files that were in use when \texttt{vi} was interrupted. It will not work if you stop \texttt{vi} yourself without saving the file; nor will it work if you save the file but then accidentally erase it with a \texttt{rm}, \texttt{cp}, or \texttt{mv} command.

The second option, \texttt{-R}, allows you to use \texttt{vi} to look at, but not change, a file. (\texttt{R} stands for "read-only.") Use this option when you want to examine an important file and guard against accidentally changing it. (An alternative is to use the \texttt{more} command to display the file.)

**WORKING WITH vi**

**How vi Uses the Screen**

The bottom line of the screen is called the \textit{COMMAND LINE}. \texttt{vi} uses the command line to display messages and commands and the rest of the screen to display the editing buffer. \texttt{vi} will display as many lines as possible; of course, this depends on the size of your screen. If you are
using the console on the IBM PC AT computer, the screen has 25 lines. Thus, \texttt{vi} will display 24 lines at a time of the editing buffer. The 25th line is the command line.

\texttt{vi} starts by displaying the beginning of the file. As you work, there are many commands that you can use to tell \texttt{vi} to display different parts of the file.

Occasionally, there will not be enough lines to fill up the screen. For example, you may edit a file that has only ten lines. It would be confusing if \texttt{vi} just left these lines blank, because the file might actually contain blank lines. Instead, \texttt{vi} marks each unused line on the screen with a tilde (\texttt{^}). So if you edit a file with ten lines, \texttt{vi} will display them on the top ten lines of the screen. Below will be 14 lines, each marked by a tilde. As you add new lines, they will take up more and more of the screen, and the tildes will disappear.

\texttt{vi} continually updates the information on the screen, so what you see is what you have. However, some terminals have characteristics that make it difficult to erase lines in the middle of the screen. With such terminals, \texttt{vi} will occasionally not be able to erase lines from the screen that you have deleted from the editing buffer. When this happens, \texttt{vi} will mark the lines with an \texttt{@} character. If you want \texttt{vi} to redraw the screen and get rid of these lines, press \texttt{<CTRL-r>}. (\texttt{r} stands for "redraw.")

In a more general sense, you can have \texttt{vi} redraw the entire screen at any time by pressing \texttt{<CTRL-l>} (the letter \texttt{l}). (\texttt{l} stands for "look.")

It is possible to put control characters into the editing buffer. For example, you may want to insert the character \texttt{CTRL-a} into a file. \texttt{vi} displays such characters on the screen as \texttt{^} followed by the symbol. If the symbol is a letter, \texttt{vi} displays it in upper case. For example, \texttt{vi} would display \texttt{CTRL-a} as \texttt{^A}. Even though it takes up two spaces on the screen, it still counts as one character.

\texttt{vi} has built-in tab settings. It assumes that these tab settings occur at every eighth place across the line. If you like, you can change the positions to suit your needs.

When you press the \texttt{<TAB>} key, you are really sending the \texttt{tab} character. This character is the same as \texttt{CTRL-i}. (In fact, you can press \texttt{<CTRL-i>} if you want, instead of \texttt{<TAB>}.) However, when you enter a tab character, \texttt{vi} does not display it as \texttt{^I}. Rather, \texttt{vi} displays enough \texttt{space} characters to bring the cursor to the next tab setting. This is handy for indenting paragraphs or making tables. Remember, even though the \texttt{tab} character is displayed as one or more \texttt{space} characters, it still counts as only one character.
The Two Modes

`MODE` refers to a particular way in which a program can work. Although it is the `vi` program that is in one mode or another, it is customary to talk as if the user is in a particular mode. For example, I may say that a certain command works only if you are in this or that mode.

`vi` can work in two different ways. That is, `vi` has two modes: COMMAND MODE and INSERT MODE. In command mode, you can use any of the `vi` or `ex` commands. In insert mode, you can add information to the editing buffer.

`vi` commands can only be used in command mode. Most commands consist of one or two characters. For example, the command to delete a line is `dd`. To use a command, simply type the characters. Except for a few commands, you do not press `<ENTER>`. For example, to delete a line, simply press `<d><d>`.

`vi` does not echo `vi` commands. For example, if you type `dd` to delete a line, you will not see "dd" on the screen. All that will happen is that a line will disappear. At first this may be a bit confusing, but you will soon get used to it.

If you make a mistake and use a bad `vi` command, `vi` will make a noise (a beep or a bell, depending on your terminal). There are no error messages for `vi` commands. However, if you make a mistake entering an `ex` command, `vi` will display an error message on the command line.

`vi` automatically starts off in command mode. From command mode you can change to insert mode whenever you want to add to the editing buffer. There are 11 different `vi` commands that you can use to change to insert mode.

While you are in insert mode, everything you type will be added to the editing buffer. As you type, `vi` will update the screen. At any time, what you see on the screen is what you have. When you are finished adding to the editing buffer, you can switch back to command mode by pressing `<ESC>`. (This is the only time that you will need to use this key with Xenix.)

Entering `ex` Commands

As I mentioned earlier, `vi` and `ex` are really different faces of the same program. This means that while you are working with `vi` you can use `ex` commands whenever you want. To enter an `ex` command, you must be in command mode. Type a colon (`:`), followed by the command, and then press `<ENTER>`.
For example, the command to substitute the string the for the string thee, is \texttt{s/thee/the/}. To enter this command from \texttt{vi} you would type:

\texttt{:s/thee/the/<ENTER>}

I told you that \texttt{vi} commands are short and simple and that \texttt{vi} does not echo them. \texttt{ex} commands, however, are more complicated, and you really have to see what you are doing as you type. For this reason, \texttt{vi} echoes \texttt{ex} commands on the command line.

As soon as you type the colon, \texttt{vi} displays it on the command line (the bottom line of the screen). As you type the rest of the command, \texttt{vi} echoes it on the same line. If you make a mistake, you can correct it with \texttt{<BACKSPACE> before you press <ENTER>}. Alternatively, you can press \texttt{<CTRL-u>} (the \texttt{kill} signal) and retype the entire command. If you decide not to enter an \texttt{ex} command after all, press \texttt{<DEL>} (the \texttt{intr} signal).

When you are entering an \texttt{ex} command, and you correct characters using \texttt{<BACKSPACE>} or \texttt{<CTRL-u>}, \texttt{vi} will move the cursor to the left, and you can retype the characters. However, the original characters will not be erased from the command line until you type over them. Don't worry—the old characters are gone, it's just that \texttt{vi} hasn't bothered to erase them from the screen.

As I describe the commands, I will put a colon in front of all \texttt{ex} commands, in order to distinguish them from \texttt{vi} commands. For example, I will refer to the substitute command above as the \texttt{:s} command. Whenever you see such a command, remember that \texttt{vi} will echo it on the command line as you type and that you must press \texttt{<ENTER> when you are finished}.

On rare occasions, you may want to change from \texttt{vi} to \texttt{ex} so you can enter a number of \texttt{ex} commands. To do so, type \texttt{Q}. This is a \texttt{vi} command that switches to \texttt{ex}. To switch back to \texttt{vi}, use the \texttt{ex} command \texttt{:vi}.

\texttt{ex} differs from \texttt{vi} in that you enter one command at a time. \texttt{ex} will display a colon as a prompt after each command. Since \texttt{ex} displays the colon, you do not need to type it yourself in front of each command. Once you have changed to \texttt{ex} you cannot use any \texttt{vi} commands.

If you are ever using \texttt{vi} and all of a sudden things are strange, and you keep seeing a colon displayed on the command line, the problem is that you have accidentally changed to \texttt{ex}. Perhaps you typed \texttt{Q} inadvertently. In any event, all you need to do is to use the \texttt{:vi} command to change back to \texttt{vi}.

Here is a summary of the commands in this section:

\begin{itemize}
  \item \texttt{Q} change from \texttt{vi} to \texttt{ex}
  \item \texttt{:vi} change from \texttt{ex} to \texttt{vi}
\end{itemize}
MANIPULATING YOUR WORK FILES

Saving Your Work and Stopping vi

There are several ways to stop vi. Most of the time, you will use the vi command ZZ. (Notice that these are upper-case letters.) When you type ZZ—you do not press <ENTER>—vi will copy the editing buffer to the current file and then stop. You will be returned to the shell.

For example, say you start vi by entering:

```
vi letter
```

vi copies the file letter to the editing buffer. You make some changes and additions and then type:

```
ZZ
```

vi copies the editing buffer to letter and then returns you to the shell.

You might wonder, Why is this command named ZZ? First, “Z” is the last letter of the alphabet, so you can remember the name because this is the last command you use during an editing session. Second, since the command takes effect as soon as you type it (without pressing <ENTER>), it was important to choose a name that would not be typed by accident.

Aside from the ZZ command, there are several ex commands that will stop vi. I will describe the most useful ones.

The first command, :x (for “exit”), is almost the same as the ZZ command. The only difference is that if you want you can specify that the editing buffer is to be copied to a different file. Simply type the name of the file after :x.

For example, if you start vi with:

```
vi letter
```

and end with:

```
:x
```

it is the same as if you used ZZ. The editing buffer is copied to the file letter, and you are returned to the shell. However, if you use:

```
:x temp
```

the editing buffer is copied to the file temp. (If it does not already exist, Xenix creates it.)

If you want to copy the editing buffer to a file without stopping vi, use the :w command. (w stands for “write.”) You can use this command to save your work without stopping vi. As with :x, you can specify a
different file name if you want. For example, the command:

: w

will copy the editing buffer to the current file *letter*, but will not stop *vi*. The command:

: w temp

will copy the editing buffer to the file *temp*.

In order to protect you from mistakes, the :w command will not copy to a file that already exists because this would erase the file. In such a case, you can override the protection mechanism by putting a ! character after the :w. For example, if the file *temp* exists, *vi* will not let you use the command:

: w temp

However, if you really want to replace *temp*, you can use:

: w! temp

The use of the ! at the end of a command name is used with several commands in order to override some sort of automatic checking.

If you want to stop *vi* without saving the contents of the editing buffer, use the :q command. (q stands for “quit.”) Again, to protect you from mistakes, *vi* will not carry out the command if you have made changes to the editing buffer and you have not saved your work. To override this check, use :q!. Be careful: :q! will stop *vi* no matter what. If you haven't saved your work, it will be lost.

Here is a summary of the commands in this section. In all cases, the file name is optional. If you do not specify it, *vi* will copy to the current file:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZZ</td>
<td>save editing buffer to current file and stop</td>
</tr>
<tr>
<td>:x [file name]</td>
<td>save editing buffer and stop</td>
</tr>
<tr>
<td>:w [file name]</td>
<td>save editing buffer; do not stop</td>
</tr>
<tr>
<td>:q [file name]</td>
<td>stop without saving editing buffer</td>
</tr>
<tr>
<td>:w! [file name]</td>
<td>same as :w but override check</td>
</tr>
<tr>
<td>:q! [file name]</td>
<td>same as :q but override check</td>
</tr>
</tbody>
</table>

**Editing More Than One File**

When you start *vi* by using the *vi* command, you can specify more than one file name as arguments. *vi* allows you to edit each file in turn. Before you move on to the next file, you must make sure to save the
When you start `vi`, the first file in the list becomes the current file. At any time, you can move on to the next file by using the `:n` command. (`n` stands for “next.”) To make sure that you do not accidentally lose information, `vi` will not execute the `:n` command if you have changed the editing buffer but have not yet saved it. If you want to override this check, you can use `:nl`.

If you forget the name of the current file, use either the `CTRL-g` or `:f` commands. `vi` will display the name of the file and other information. These two commands are equivalent. The only difference is that `CTRL-g` is a `vi` command. It is a little faster to use a `vi` command because you do not have to press `<ENTER>`.

You can change the name of the current file by specifying a new file name with the `:f` command. For example, to change the name of the current file to `newfile`,

```
   enter: :f newfile
```

This does not affect the original file at all. It just means that if you save the editing buffer to the current file, it will go to this new file.

If you want to see the names of all the files in the argument list, use the `:args` command. The current file will be enclosed in square brackets. For example, say you start `vi` by entering:

```
   vi file1 file2 file3 file4
```

and that `file3` is now the current file. If you enter:

```
   :args
```

you will see:

```
   file1 file2 [file3] file4
```

If you want to start editing again at the beginning of the list, use the `:rew` command. (`rew` stands for “rewind.”) This command makes the first file the current file. For example, if you are working with the files in the last example, and you enter:

```
   :rew
   :args
```

you will see:

```
   [file1] file2 file3 file4
```

Once again, `vi` will not execute the command if you have changed the editing buffer but not saved it. To override this check, use `:rew!`. 
If you want to edit a file that is not in the argument list you have two choices. First you can use the :n command with a new list of files. This list replaces the argument list. Second, you can use the :e command with the name of one file. (:e stands for "edit.") The :e command will change the file you are editing, but it will not change the argument list; you can come back to these files later by using :rew. As with the other commands that change the contents of the editing buffer, you can use a ! to override the automatic checking.

Here is a summary of the commands in this section:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>:n</td>
<td>edit next file in argument list</td>
</tr>
<tr>
<td>CTRL-g</td>
<td>display name of current file</td>
</tr>
<tr>
<td>:f</td>
<td>same as CTRL-g</td>
</tr>
<tr>
<td>:args</td>
<td>display argument list</td>
</tr>
<tr>
<td>:rew</td>
<td>edit first file in argument list</td>
</tr>
<tr>
<td>:n [file name...]</td>
<td>edit new file; change argument list</td>
</tr>
<tr>
<td>:e [file name]</td>
<td>edit new file; same argument list</td>
</tr>
<tr>
<td>:rew!</td>
<td>same as :rew but override check</td>
</tr>
<tr>
<td>:n!</td>
<td>same as :n but override check</td>
</tr>
<tr>
<td>:e!</td>
<td>same as :e but override check</td>
</tr>
</tbody>
</table>
CHAPTER 11
THE vi EDITOR—DISPLAYING TEXT FILES AND MOVING THE CURSOR

INTRODUCTION
MOVING THE CURSOR—1
USING PATTERNS
MOVING THE CURSOR—2
MORE WAYS TO CONTROL THE SCREEN DISPLAY
INTRODUCTION

The vi editor uses the full screen to display as many lines of your file as possible. However, you will usually not be able to see all the lines at once. vi has many commands to allow you to move from one part of the file to another. The commands involve moving the cursor to the place that you want to display or modify. This chapter introduces you to the vi cursor and teaches you how to control it.

New Words in This Chapter

CURRENT LINE
the line on which the cursor lies

ABSOLUTE MOVE COMMAND
a command that moves the cursor to a specific line

MOVING THE CURSOR

The Importance of the Cursor

As you remember, the cursor is the symbol that marks your position on the screen. With the IBM PC AT Computer, the cursor is a blinking underscore character. At any time, the line on which the cursor lies is called the CURRENT LINE.

When you use vi, the cursor marks the position at which you can modify the file. For example, if you want to add a word to a sentence, you would first move the cursor to the position where you want to add the word. You would then change to insert mode and add the word.

Although it is the cursor that moves and that has a particular position, we often speak of the user as being in that position. For example, I may say that to delete a word, you move to that word and then use the dw command.

vi makes sure that the cursor is always somewhere on the screen. For example, if you move 100 lines farther into the editing buffer, vi will automatically update the display to keep the cursor on the screen. Thus, you can display any part of the editing buffer by moving to that position.
**Displaying Text Files and Moving the Cursor**

`vi` has many different ways to move the cursor. This gives you great control over your work. You can easily move backward, forward, left, and right. You can move to the next word, the previous word, the next paragraph, the previous section, and so on.

The cursor commands are simple and easy to use. The next few sections will show you how they work. As you read, it is best if you try out the commands on your terminal. To do so, you will need to have a file to edit. Use the file that holds the table of conversion information for the `units` program: `/usr/lib/unittab`.

Use the command:

```
vi -R /usr/lib/unittab
```

Notice the use of the `-R` option ("read-only") to make sure that you do not accidentally modify the file. Log in to Xenix, enter this command, and then continue reading.

**Moving in Big Jumps**

`vi` displays the editing buffer by starting with the first line at the top of the screen. Thus, it is convenient to refer to moving toward the end of the editing buffer as moving "down" or "forward." Similarly, moving toward the beginning of the editing buffer can be thought of as moving "up" or "backward."

`vi` has four commands that are named in this way. They are:

- `CTRL-f` move one screenful down (forward)
- `CTRL-b` move one screenful up (backward)
- `CTRL-d` move a half screenful down
- `CTRL-u` move a half screenful up

Try these commands. These are the commands to use when you want to page through the editing buffer in large or medium jumps.

**Moving Using Line Numbers**

`vi` gives each line of the editing buffer an internal number. You can move to any line by telling `vi` to go to that line number. Of course, to do this, you need to see the line numbers. You can tell `vi` to display line numbers by using the `ex` command:
Moving the Cursor

:set number

Enter this command now. (Remember to press <ENTER>.) You will see line numbers along the left side of the screen. Move up and down in the file. Notice how the line numbers show you where you are in the file. If you want vi to stop displaying line numbers, use:

:set nonumber

Most of the time it is convenient to see line numbers. Many people routinely use :set number. My suggestion is that you always display line numbers. Later, you will learn how to customize your working environment. At that time, you can specify that the :set number command is to be executed automatically whenever you start vi.

It is important to realize that the line numbers are not really part of the editing buffer; vi simply displays them for your convenience. The line numbers always run consecutively from 1 through to the last line in the file. When you insert or delete lines, vi automatically updates the line numbers. For example, if you insert new lines after line 10, all the lines from 11 to the end of the editing buffer are renumbered to make room.

To go to a particular line in the file, type the line number followed by G. (G—notice it is upper case—stands for “go to.”) For example, to go to line 100, use:

100G

To go to the beginning of the file, use:

1G

vi recognizes the symbol $ as standing for the last line in the file. This means that you can refer to the last line even if you don’t know what number it is. For example, to go to the last line in the file, use:

$G

As a convenience, vi also lets you use G<ENTER> to go to the last line. You may find this easier to type than $G.

With some commands, you can use 0 (zero) as a line number. In such cases, 0 is used to mean “before line 1.” For example, in order to copy a line to the beginning of the editing buffer, you would copy it to line 0.

Here is a summary of the commands in this section:

line numberG  go to the line with a specified number
G<ENTER>     go to the last line
USING PATTERNS

Searching for a Pattern

To go to a line that contains a particular pattern, type slash (/), followed by the pattern, and then press <ENTER>. As you type, vi echoes the slash and the pattern on the command line (the same as when you type an ex command). If necessary, you can make corrections before you press <ENTER> by using <BACKSPACE> or <CTRL-u>. To cancel the whole thing, press <DEL>.

Remember, as with ex commands, you must press <ENTER> after you type the command. In order to make the examples more readable, I will not put <ENTER> after each one.

Here is an example: To go to the next line that contains the pattern dollar,

```
enter: /dollar
```

Try this example now. (Remember, you should be editing the file /usr/lib/unittab.)

When you use / to search for a pattern, vi starts from the current line and moves forward (down) in the editing buffer. If none of the lines up to the end of the editing buffer contain the pattern, vi will wrap around and continue to search forward from line 1. If vi gets back to the current line without finding the pattern, you will see the message:

```
pattern not found
```

If you use a ? (question mark) instead of /, vi will search backward (up). In this case, if the pattern is not found anywhere up to line 1, vi will wrap around and continue to search backward from the last line.

Try this example:

```
?dollar
```

It is often handy to search repeatedly for the same pattern. For example, you may search for the pattern dollar, but vi may not find the one you want, so you will want to look for the next one.

To repeat the last / or ? command, type n. (n stands for "next.") If you want to repeat the last command but in the reverse direction, use N.

For example, say you search backward for the pattern dollar. You can press n repeatedly to continue the search forward. Take a few moments now and experiment with backward and forward searches and with n and N to make sure you understand.

If you enter / or ? by itself, vi will search for the last pattern in the direction you specify. For example, say you search forward for the
pattern **hello** by entering:

```
/hello
```

If you enter /, *vi* will repeat the command. If you enter ?, *vi* will search backward for the same pattern.

Make sure you understand the difference between / and ? by themselves, and n and N. / and ? always search forward and backward, respectively. The direction that n and N search depends on the previous command.

Sometimes you may wish to find a pattern and then move up or down from there. You can do so with the following commands:

```
/pattern/+n
/pattern/-n
?pattern?+n
?pattern?-n
```

For example, to go to the fifth line past the next occurrence of **dollar**, use:

```
/dollar/+5
```

To go to the third line before the last occurrence of **dollar**, use:

```
?dollar?-3
```

Here is a summary of the commands in this section. Take a moment to make sure you understand how each one works. **pattern** stands for a regular expression. n stands for a number.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/pattern</td>
<td>search forward: go to next line with pattern</td>
</tr>
<tr>
<td>?pattern</td>
<td>search backward: go to next line with pattern</td>
</tr>
<tr>
<td>/</td>
<td>search forward: for previous pattern</td>
</tr>
<tr>
<td>?</td>
<td>search backward: for previous pattern</td>
</tr>
<tr>
<td>n</td>
<td>same direction: repeat last / or ? command</td>
</tr>
<tr>
<td>N</td>
<td>reverse direction: repeat last / or ? command</td>
</tr>
<tr>
<td>/pattern/+n</td>
<td>search forward: go to nth line after pattern</td>
</tr>
<tr>
<td>?pattern?-n</td>
<td>search forward: go to nth line before pattern</td>
</tr>
<tr>
<td>?pattern?+n</td>
<td>search backward: go to nth line after pattern</td>
</tr>
<tr>
<td>?pattern?-n</td>
<td>search backward: go to nth line before pattern</td>
</tr>
</tbody>
</table>
Using Regular Expressions with vi

With vi, you can use a regular expression to specify a pattern. In particular, you can use regular expressions with the / and ? search commands. The rules for forming regular expressions are similar to those used by Xenix in general. In this section, I will describe the rules that apply to vi. You may want to review the section in Chapter 9 that introduced regular expressions before you continue.

The period (.) matches any single character except a newline. For example, to search for the next occurrence of a, followed by any character, followed by c, enter: /a.c

Here are examples of strings that this command would match:

abc abc aHc a%c

The * character matches zero or more occurrences of the preceding character. For example, to search for the string consisting of a, followed by zero or more occurrences of b, followed by c, use:

/ab*c

Here are examples of strings that this command would match:

ac abc abbbbc aHELLOc

The ^ character matches the beginning of a line. The $ character matches the end of a line. For example, to search for the word hello you would use:

/hello

However, to search for hello only at the beginning of a line, use:

/^hello

To search for hello at the end of a line, use:

/hello$

To search for a line that contains only the string hello, use:

/^hello$

Think of ^ and $ as being invisible characters at the beginning and end of every line.

Analogously, the two characters \ match the beginning of a word and the two characters \> match the end of a word. For example, to search for the next word that starts with he, use:
Here are some strings that this would match:

he hello heXXXXX12345

To search for the next word that ends with llo, use:

/llo\>

Here are some strings that this would match:

llo hello pollo

To search for the string hello as a word, use:

/\<hello\>

This command would find hello but not shello. The command:

/\<hello\>

on the other hand, would find both. Think of \(<\text{ and }\)\> as being invisible characters at the beginning and end of every word. (*\text{vi}\* considers a “word” to be a group of consecutive upper- or lower-case letters, numbers, or underscore characters.)

You can use the [ and ] characters to enclose a set of characters. This construction matches any one member of the set. For example, to search for either hello or Hello, use:

/\[hH\]ello

Within [ and ], you can use the - character to indicate a range of letters or numbers. For example, to search for the next upper-case letter, use:

/\[A-Z]\]

To search for a string consisting of one upper-case letter followed by zero or more lower-case letters, use:

/\[A-Z\][a-z]*/

Notice that the * refers only to the preceding character; [a-z] represents one character. This last example would match the following strings:

Aaaaaa Hello xxxxXyyyyy worD

In the third string, the regular expression matches Xyyyyy (one upper-case letter followed by five lower-case letters). In the fourth string, the regular expression matches D (one upper-case letter followed by zero lower-case letters). If you want to match the pattern as a complete word, you would use:
Sometimes you want to match a character that is not one of a set. In this case, put a ^ immediately after the [. For example, to search for the next word that starts with any character except a, b, or c, use:

```
/[^abc]/
```

Take care not to confuse a ^ at the beginning of a regular expression (match the beginning of a line) with a ^ immediately after a [. They have two different meanings.

If you want to match a character that has a special meaning, you must quote it by preceding it with \. (a backslash). For example, to search for the string the.dot, use:

```
/the\.dot
```

Sometimes it can get tricky and you have to think carefully. For example, to search for zero or more lower-case letters, followed by *, followed by zero or more numbers, use:

```
/[a-z]\*[0-9]*
```

Also, remember that the < and > have special meanings only when preceded by \. (The same goes for ( and ), which have a special meaning, explained later.) For example, to search for 85<110, use:

```
/85\<110
```

I know this is confusing, but with practice you will get used to it.

Here is a summary of the rules described in this section. These characters have special meanings only when used in regular expressions to specify a pattern.

Regular expressions can be used to specify patterns in vi. Within regular expressions, the following characters have special meanings:

- . matches any one character except a new\line
- * matches zero or more of the preceding characters
- ^ matches the beginning of a line
- $ matches the end of a line
- \< matches the beginning of a word
- \> matches the end of a word
- [ ] matches one of the enclosed characters
- [^] matches any character that is not enclosed

Inside [ and ], you can use - to indicate a range of single letters or digits. \ indicates that the following character is to be taken literally, except for \<, \>, \(, and \).
MOVING THE CURSOR—2

Moving in Medium Jumps

There are three commands that move the cursor to the beginning of the top, middle, and bottom lines of the screen. They are:

- **H**: move cursor to top line of screen
- **M**: move cursor to middle line of screen
- **L**: move cursor to bottom line of screen

H stands for “high”; M stands for “middle”; and L stands for “low.”

vi has commands that allow you to move the cursor in terms of paragraphs, sentences, and words.

- `{`: move cursor forward to beginning of paragraph
- `}`: move cursor backward to beginning of paragraph
- `)`: move cursor forward to beginning of sentence
- `(`: move cursor backward to beginning of sentence
- **w**: move cursor forward to beginning of next word
- **e**: move cursor forward to last character in word
- **b**: move cursor backward to beginning of word

w stands for “word”; e stands for “end of word”; and b stands for “backward.”

The paragraph commands consider a paragraph to end with a blank line. The sentence commands consider a sentence to end with either a period, exclamation point, or question mark, followed by two space characters or a newline. The word commands consider a word to end with any character that is not a letter, number, or underscore.

Take a moment to try out these commands to make sure that you understand how they work. In particular, remember the e command. It will come in handy when you want to add letters to the end of a word.

You may have noticed that the w, e, and b commands stop at every punctuation character. If this is inconvenient, you can use the W, E, and B commands. They work the same way except that they consider words to be ended only by a space or a newline. Think of the upper-case letters as standing for bigger words.

- **W**: same as w; ignore punctuation
- **E**: same as e; ignore punctuation
- **B**: same as b; ignore punctuation
Moving in Small Jumps

There are a few commands that are particularly well suited to moving within a small area.

0 (the number zero) moves the cursor to the beginning of the current line. \^ moves the cursor to the first character in the current line that is not a space or a tab. (If the current line does not start with a space or tab, these two command are equivalent.) The \^ command is handy when you are working with lines that are indented.

$ moves the cursor to the end of the current line.

If you want to move to a particular column in the current line, type the number, followed by \| (the vertical bar). (Think of \| as referring to a column, straight up and down.) For example, to move to column 25 of the current line, use:

25\|

You can also move forward to the next occurrence of a particular character in the current line. Type \f followed by the character. (\f stands for “find.”) For example, to go to the the next semicolon on the current line, use:

\f;

This command will only search up to the end of the current line. If it cannot find the character, \v will beep to indicate an error.

The \F command is the same as \f except that it searches backward on the current line. The \t and \T commands are similar to \f and \F except that they leave the cursor one position before the character (in the direction of the search).

For example, say that the current line is:

```
abcdefghijklmnopqrstuvwxyz
```

and the cursor is at the l (between the m and n). Here are examples of the effects of the different commands:

- fx move cursor right to x
- Fc move cursor left to c
- tx move cursor right to w
- Tc move cursor left to d

\v remembers that last \f, \F, \t, or \T command. You can use ; (a semicolon) to repeat the command. You can use , (a comma) to repeat the command in the opposite direction. (This is analogous to using n and N to repeat the / and ? search commands.)

Here is a summary of the commands in this section. Take a moment
now to make sure you understand how each one works. char means any character. \( n \) means a number.

| 0 | move cursor to beginning of current line |
| ^ | move cursor to first non-space/tab in current line |
| $ | move cursor to end of current line |
| n | move cursor to \( n \)th column of current line |
| tchar | move cursor right to next occurrence of character |
| Fchar | move cursor left to next occurrence of character |
| tchar | move cursor right to position before character |
| Tchar | move cursor left to position before character |
| ; | same direction: repeat last f, F, t, T command |
| \, | reverse direction: repeat last f, F, t, T command |

Moving in Very Small Jumps

There are a variety of different ways that you can move the cursor very short distances.

+ moves the cursor down to the beginning of the next line. - moves the cursor up to the beginning of the previous line. Both + and - position the cursor at the first character of the line that is not a space or tab. This makes it easy to work with lines that are indented. If you are using indented lines and you want to move the cursor to the absolute beginning of a line (before the space or tab characters), use the 0 (zero) command.

+ and - are particularly easy to use with the IBM PC AT computer because there are <+> and <-> keys on the right side of the numeric keypad. As a convenience with other terminals, you can use <ENTER> by itself to mean the same thing as +. In other words, you can press <ENTER> repeatedly (in command mode) to move through the editing buffer one line at a time.

To move the cursor one position at a time, use the cursor control keys. (On the IBM PC AT, these are the keys with the arrows on the numeric keypad.) Each of these keys moves the cursor one position in the direction of the arrow.

When you use <CURSOR-UP> (or <CURSOR-DOWN>), vl will move the cursor to the same column in the line above (or below). If the line is not long enough, vl will move the cursor to the end of the line. For example, if you are in column 25 of the current line, <CURSOR-UP>
will move you to column 25 of the line above. If this line is less than 25 characters long, the cursor will be at the end of the line.

As a convenience, there are two alternate ways to specify each of these four cursor movements:

<table>
<thead>
<tr>
<th>Command</th>
<th>Alternative Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;CURSOR-RIGHT&gt;</code></td>
<td>either <code>&lt;SPACE&gt;</code> or <code>j</code></td>
</tr>
<tr>
<td><code>&lt;CURSOR-LEFT&gt;</code></td>
<td>either <code>&lt;BACKSPACE&gt;</code> or <code>h</code></td>
</tr>
<tr>
<td><code>&lt;CURSOR-UP&gt;</code></td>
<td>either <code>&lt;CTRL-p&gt;</code> or <code>k</code></td>
</tr>
<tr>
<td><code>&lt;CURSOR-DOWN&gt;</code></td>
<td>either <code>&lt;CTRL-n&gt;</code> or <code>l</code></td>
</tr>
</tbody>
</table>

The `<CTRL-p>` stands for "previous line." The `<CTRL-n>` stands for "next line." Remember, these are vi commands, to be used in command mode. In particular, the `<SPACE>` and `<BACKSPACE>` keys have their regular effect in insert mode or when you are typing an ex command or search (/ or ?) command.

You might wonder why the letters `h`, `j`, `k`, and `l` were chosen as alternate ways to move the cursor. The answer is that some of the terminals that were used when vi was developed had the cursor-control symbols on these keys. The use of these keys has simply been continued.

There is no need for you to learn three different ways to move the cursor. Try the different keys on your terminal and select one set that you like the best. If you are using a terminal with easily accessible cursor-control keys, you may want to use them. If you are a touch typist, you may want to use the `h`, `j`, `k`, and `l` keys so you won't have to move your fingers off the keyboard.

Here is a summary of the commands in this section. Take a moment now to make sure you understand how each one works.

+ move cursor to first non-space/tab of previous line
- move cursor to first non-space/tab of next line

ENTER same as +
CURSOR-RIGHT move cursor one position right
CURSOR-LEFT move cursor one position left
CURSOR-UP move cursor one position up
CURSOR-DOWN move cursor one position down
SPACE same as `<CURSOR-RIGHT>`
BACKSPACE same as `<CURSOR-LEFT>`
CTRL-p same as `<CURSOR-UP>`
CTRL-n same as `<CURSOR-DOWN>`
l same as `<CURSOR-UP>`
h same as `<CURSOR-LEFT>`
k same as `<CURSOR-UP>`
j same as `<CURSOR-DOWN>`
MORE WAYS TO CONTROL THE SCREEN DISPLAY

Changing the Screen Position of the Current Line

As you know, you can have vi redraw the screen at any time by using <CTRL-D>. Using the z command, you can have vi redraw the screen with the current line at a particular position.

- **z<ENTER>** redraw screen: current line at top
- **z.** redraw screen: current line in middle
- **z-** redraw screen: current line at bottom

(Notice that you only press <ENTER> with the first form of the z command.)

Here is how you might use this command: If you want to display a particular line at the top of the screen, move the cursor to that line and then type z<ENTER>.

If you want, you can have vi redraw the screen with a particular line at the top, middle, or bottom. Simply put the line number in front of the z. For example, the following commands redraw the screen with line 204 at the top, middle, and bottom, respectively:

- **204z<ENTER>**
- **204z.**
- **204z-**

You can also put a search command (/ or ?) in front of the z command. Remember, the pattern can be any regular expression.

Here are two examples: The first searches forward for the first line that begins with the string **hello** and redraws the screen with that line in the middle. The second command does the same thing but searches backward.

- **/^hello/z.**
- **?^hello?z.**

There are two things to mention here. First, you must put a second / or ? after the pattern to separate it from the z. Second, as soon as you type / or ?, vi will start echoing what you type on the command line. In this case, you will have to press <ENTER> to enter the command, even if you use z. or z-.

Here is a summary of the commands in this section. Take a moment to make sure you understand how each one works. line refers to a line number, and pattern refers to a string or regular expression.
Displaying Text Files and Moving the Cursor

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>z&lt;ENTER&gt;</td>
<td>redraw screen: current line at top</td>
</tr>
<tr>
<td>z-</td>
<td>redraw screen: current line at middle</td>
</tr>
<tr>
<td>z.</td>
<td>redraw screen: current line at bottom</td>
</tr>
<tr>
<td>linez&lt;ENTER&gt;</td>
<td>redraw screen: specified line at top</td>
</tr>
<tr>
<td>linez-</td>
<td>redraw screen: specified line at middle</td>
</tr>
<tr>
<td>linez.</td>
<td>redraw screen: specified line at bottom</td>
</tr>
<tr>
<td>/pattern/z &lt;ENTER&gt;</td>
<td>redraw screen: line with pattern at top</td>
</tr>
<tr>
<td>/pattern/z-&lt;ENTER&gt;</td>
<td>redraw screen: line with pattern at middle</td>
</tr>
<tr>
<td>/pattern/z.&lt;ENTER&gt;</td>
<td>redraw screen: line with pattern at bottom</td>
</tr>
<tr>
<td>?pattern?z&lt;ENTER&gt;</td>
<td>redraw screen: line with pattern at top</td>
</tr>
<tr>
<td>?pattern?z-&lt;ENTER&gt;</td>
<td>redraw screen: line with pattern at middle</td>
</tr>
<tr>
<td>?pattern?z.&lt;ENTER&gt;</td>
<td>redraw screen: line with pattern at bottom</td>
</tr>
</tbody>
</table>

Returning and Marking

A command that moves the cursor to a specific line is called an ABSOLUTE MOVE COMMAND. So far, you have learned three absolute move commands: G, /, and ?, and their variations.

Whenever you use one of these commands, vi makes a note of the cursor position before the move. This is so that you can easily return to the line from which you moved.

To move the cursor to the position it had the last time you used an absolute move command, use "" (two back-quotes).

For example, say that you are editing a long document and you suddenly remember that you want to change something in the first line. Use 1G to move the cursor to line 1. After you have made the change, use "" to return to your previous position.

The "" command will return to the exact position on the line. If you want to return to the beginning of that line, use ' ' (two single-quotes).
The "" command returns to the first non-space, non-tab character in the line.

For example, say that the current line is:

this is the current line

and the cursor is at the c in current. You use /hello to move to the next line with the string hello. If you type '', vl will move the cursor back to the c in current. If you type ' ', vl will move the cursor to the t in this.

One drawback of the "" and '"' commands is that they only return to the last place from which you used an absolute move command. Occasionally, you will want to mark a place in the editing buffer and return to it at your convenience. To do this, use the m command. (m stands for "mark.")

When you mark a place in the editing buffer, you give it a name. The name must be one lower-case letter; any letter from a to z. Thus, you can mark up to 26 places (although you will rarely need more than one). To mark a place, move the cursor to it and type m followed by the name. For example, to give a particular place the name a, type ma.

vl does not display the name of the mark on the screen. So think of the m command as letting you put up to 26 invisible marks in the editing buffer.

To move the cursor to a mark, type ' (a back-quote) followed by the name of the mark. For example, to move the cursor to the position that you marked with ma, type 'a.

If you use ' (a single-quote) instead of ' (a back-quote), vl will move the cursor to the first non-space, non-tab character on the line in which the mark appears. This is analogous to the distinction between "" and '"'.

For example, say that the current line is:

this is the current line

and the cursor is at the c in current. You type ma to mark the place with the name a. Anytime later, you can type 'a to move the c in current. If you type 'a, vl will move the cursor to the t in this.

Once you establish a mark, it persists until you make a new one with the same name. Note: All marks are lost when you stop vl or change to a new file.

Since the "", '"', ' and ' commands move the cursor to a specific line, they are absolute move commands. Thus, to be complete, we can say that vl remembers the position from which you used the last G, /, ?, '"', '"', ' or ' command.

Here is a summary of the commands in this section. Take a moment now to make sure you understand how each one works. Ichar stands for any single lower-case character.
The absolute move commands are \texttt{G}, \texttt{/}, \texttt{?}, \\texttt{"}, \\texttt{',} and \\texttt{'.}

\begin{itemize}
  \item \texttt{"} move cursor to place of last absolute move command
  \item \texttt{"}: but move to first non-\texttt{space/tab} on line
  \item \texttt{mlchar} mark place in editing buffer with specified name
  \item \texttt{\textbackslash lchar} move cursor to specified place
  \item \texttt{\textbackslash lchar} ': but move to first non-\texttt{space/tab} on line
\end{itemize}

Repeating \texttt{vi} Commands

There are two ways that you can execute \texttt{vi} commands repeatedly. First, you can precede most \texttt{vi} commands with a number to execute the command that number of times. For example, the command \texttt{w} moves the cursor forward to the beginning of the next word; \texttt{5w} moves forward 5 words. Similarly, \texttt{(} moves backward to the beginning of a sentence; \texttt{10(} moves backward 10 sentences.

The only time you cannot repeat a command in this way is when it would not make sense. For example, you cannot put a number in front of the \texttt{"} command; and a number in front of \texttt{G} or \texttt{z} would mean a line number.

The second way to execute a \texttt{vi} command repeatedly works for those commands that are one character long. Most terminals are designed so that if you hold down a key, it sends the signal for its character repeatedly. You can take advantage of this feature to repeat most of the cursor movement commands. For example, you can hold down the \texttt{<b>} key to move backward, word by word. Or, you can hold down a cursor-control key to move in a particular direction.

Occasionally, you may run into a problem with your terminal sending signals too fast when you hold down a key. With the IBM PC AT, this may happen when you hold down one of the cursor-control keys. At such times, \texttt{vi} will beep and erase part of the current line. If this happens, simply type \texttt{U} to restore the line. (This command is discussed in the next chapter.)

The two ways to repeat commands are simple, but you must build the habit of using them. For example, it is better to type \texttt{5b} or to hold down the \texttt{b} key than to press \texttt{<CURSOR-LEFT>} many times.

Just as important, you should teach yourself to move the cursor as efficiently as you can. When you have to move to a new position, try to make a large jump first to get near the new place, and then small jumps to zero in.
For example, if you are near the bottom left-hand corner of the screen, and you want to move to a word near the top right-hand corner, it would be very slow to use <CURSOR-UP> and <CURSOR-RIGHT>. Rather, type H$ to get to the end of the top line, and then use b to back up to the word you want.

Take a few moments now to practice repeating the commands that you have learned in this chapter. If you find yourself constantly depending on the cursor-control keys, it is a good indication that you should take some time to practice the other commands.
CHAPTER 12
THE vi EDITOR—CREATING AND MODIFYING TEXT FILES

INTRODUCTION
USING THE PREVIOUS COMMAND
INSERTING
REPLACING, CHANGING, AND DELETING
USING THE BUFFERS
INTRODUCTION

The previous two chapters showed you how to start and stop `vi` and how to display your file by moving the cursor. In this chapter, you will learn how to add to and modify the information in the editing buffer. By the time you finish this chapter, you will be able to create and change your own text files.

New Words in This Chapter

DELETE BUFFER
one of nine temporary storage areas, used by `vi` to hold the nine most recent deletions that involved a line, a sentence, or anything longer; the delete buffers are numbered 1 through 9

NAMED BUFFER
one of 26 temporary storage areas available to the user; the named buffers are known by the letters of the alphabet, a through z

UNNAMED BUFFER
a temporary storage area used by `vi` to hold the most recent deletion

Before You Start

In order to create your own text files, you start `vi` and enter insert mode. You then insert information into the editing buffer. You change back and forth from insert mode to command mode as the need arises. When you are finished, you save the editing buffer into a file. If the file does not exist, Xenix will create it for you.

As you read this chapter, you will need to practice the commands to insert into the editing buffer and to modify information already in the buffer. To do this, start `vi` with the name of a file called `temp` that you will use as a temporary work file. (If you already happen to have a file named `temp`, use a different name.) Enter this command:

```
vi temp
```

before you go on to the next section. As well, have `vi` display the line numbers so you can better understand what is going on.

Enter: `:set number`
154 Creating and Modifying Text Files

As I explain each command, try it out to make sure you understand how it works. When you are finished, you can stop vi and either save the file (use ZZ) or discard the file (use :q!).

USING THE PREVIOUS COMMAND

Undoing and Repeating

vi has a particularly useful feature: At any time, you can reverse the effect of the last command that modified the contents of the editing buffer by using the u command. (u stands for “undo.”)

For example, say that you have mistakenly deleted the first 1000 lines of the editing buffer. Simply type u and they will reappear. Or, say that you insert ten lines into the editing buffer, and then you decide you don’t really want them. Type u, and they will disappear.

Be careful: The u command only undoes the last command to modify the editing buffer. If you make a bad mistake, make sure you type u immediately. Do not get all excited and accidentally enter another command that modifies the editing buffer.

Since u itself modifies the editing buffer, a second u undoes the effects of the first. For example, say you delete 1000 lines and then type u to recover them. If you type u again, they will disappear; type u again, and they reappear; and so on.

A similar command, U, will reverse all the modifications to the current line that have taken place since the cursor was moved to the line.

For example, say that you move the cursor to line 100 and then make all sorts of changes. If you type U, vi will restore line 100 to what it looked like before the changes.

Be careful: The U command only undoes changes to the current line. If you make changes to line 100 and then move the cursor to line 99, U will not be able to restore line 100. Thus, if you accidentally ruin the current line, make sure you type U before you move the cursor.

The u command undoes the last command that modified the editing buffer. The . (period) command does the opposite: It repeats the last command that modified the editing buffer.

For example, after you use the command to delete a word (see below), you can type a period to delete another word. It is occasionally useful to type a period repeatedly to execute a command over and over.
You can type a number before the period to repeat the command that many times. For example, type 10. to repeat a command 10 times.

Here is a summary of the commands in this section. Take a few moments to make sure you understand how each one works.

- **u** undo last command that modified the editing buffer
- **U** restore current line
- **.** repeat last command that modified the editing buffer

**INSERTING**

**Inserting Information into the Editing Buffer**

There are 12 vi commands that you can use to change to insert mode. In this section, you will learn six commands that will allow you to insert information anywhere in the editing buffer. The other six commands allow you to replace information that is already in the buffer.

Think of being in insert mode as having an opening into the editing buffer. Everything you type is inserted into this opening. All the characters to the right are moved over to make room; nothing is lost. If you create new lines, vi automatically renumbers all the following lines. As you type, vi continually updates the screen, so what you see is what you have. When you want to change from insert mode back to command mode, press <ESC>.

Occasionally, you may forget whether you are in command mode or insert mode; usually, there is no way of telling by looking at the screen. If this happens, press <ESC> until your terminal beeps. You will be in command mode.

At the beginning, this may happen a lot and you will curse the designers of vi: “Why couldn’t they display something on the command line to tell me when I am in insert mode?” Don’t worry: Once you get used to vi, you will know what mode you are in without even thinking about it.

The commands in this section differ only as to where they open the editing buffer. Once you are in insert mode, the commands are all the same.

The i command opens the editing buffer directly after the position of the cursor. The a command opens the editing buffer directly before the position of the cursor. For example, say the current line is:
this is the current line

and the cursor is positioned at the h in the. If you use the I command to change to insert mode, everything you type will be inserted between the h and the e. If you use the a command to change to insert mode, everything you type will be inserted between the t and the h. (I stands for "insert," and a stands for "append.")

The I command (upper-case letter "i") will open the editing buffer before the first non-space, non-tab character in the line. The A command will open the editing buffer after the last character of the line.

For example, say that the current line is:

this is the current line

If you use the I command to change to insert mode, everything you type will be inserted before the t in this. If you use the A command to change to insert mode, everything you type will be inserted after the e in line.

The o command opens a whole new line above the current line. The O command opens a whole new line below the current line. (o stands for "open.")

For example, say that the current line is line 35. If you use the o command to change to insert mode, vi will create an empty line and place the cursor at the beginning of it. Everything you type will be inserted into this line. All the lines below will be renumbered as necessary. For instance, the old current line will become line 36. If you insert more than one line, vi will renumber the lines below as often as necessary.

If the current line is line 35 and you use the O command to change to insert mode, it works the same way except that the new lines will be created after line 35, rather than before it.

You might ask: Why are there so many ways to change to insert mode? Couldn't you get by with just one such command, say the I command? The answer is that most of the time you could get by with just the I command. However, it would mean that you would have to move the cursor a lot more often to position it to just the right place. As well, without the a and A commands, it would be difficult to insert information at the end of a line.

When you start work with a new file, vi automatically places the cursor at the beginning of an empty line 1. To start work, type I to change to insert mode. Type as much as you want. When you are finished or you want to stop to make a change, press <ESC> to change to command mode.

As with other vi commands, you can execute the insertion commands repeatedly by preceding them with a number. With these
commands, `vi` will repeat their insertion as many times as you specify, if it fits on line.

For example, if you type `5iab<ESC>`, it is the same as `iabababab<ESC>`. Thus, an easy way to insert 20 periods is `20i.<ESC>`.

Here is a particular combination of commands that is so handy that you should memorize it outright. Whenever you want to add an `s` to a word, move the cursor to anywhere in the word, and type `eas<ESC>`. The `e` moves the cursor to the last letter of the word. The `a` puts you in insert mode following this letter. The `s<ESC>` inserts the `s` and changes back to command mode.

Here is a summary of the commands in this section. Take a few moments to make sure you understand how each one works.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>I</code></td>
<td>change to insert mode: insert after cursor position</td>
</tr>
<tr>
<td><code>I</code></td>
<td>change to insert mode: insert at end of current line.</td>
</tr>
<tr>
<td><code>a</code></td>
<td>change to insert mode: insert before cursor position</td>
</tr>
<tr>
<td><code>A</code></td>
<td>change to insert mode: insert at beginning of current line</td>
</tr>
<tr>
<td><code>o</code></td>
<td>change to insert mode: open new line above current line</td>
</tr>
<tr>
<td><code>O</code></td>
<td>change to insert mode: open new line below current line</td>
</tr>
</tbody>
</table>

**Correcting Mistakes in Insert Mode**

If you are typing in insert mode, you can make corrections before you press `<ESC>` by using the following commands:

- `<BACKSPACE>` erase last character
- `<CTRL-w>` erase last word
- `<CTRL-u>` erase to beginning of line

For example, say you are inserting the sentence `Hello to you` but you accidentally type `Hello to me`. Before you change back to command mode by pressing `<ESC>`, you can correct the last word by typing `<CTRL-w>` and then `you`. So the whole thing would look like:

`Hello to me<CTRL-w>you`
Here is an important point that you must understand. When you type in insert mode, *vi* makes room for the characters and echoes them on the screen. However, nothing is actually inserted into the editing buffer until you press <ESC>.

If you correct mistakes, *vi* moves the cursor to the left but does not erase the characters from the screen; it looks as if they are still there. Don’t worry, the characters really are erased, as you will see when you press <ESC>.

As you may have noticed, the rules for typing in insert mode are similar to those when you type an ex command (after the :) and a search command (after the / or ?).

### Inserting Characters That Have Special Meanings

Occasionally, you may want to insert a character that has a special meaning. If so, preface the character with <CTRL-v>. This tells *vi* to insert the next character literally and not to interpret it in a special way. This is the way to insert characters such as `CTRL-w` or `newline`.

For example, to insert the `newline` character, type `<CTRL-v>` <ENTER>. To insert `CTRL-w`, type `<CTRL-v>` `<CTRL-w>`. *vi* displays control characters on the screen using the `^` notation. For example, `CTRL-w` is displayed as `^W`. Even though it takes up two positions on the screen, it still counts as one character.

Note: The `newline` and `erase` characters are `CTRL-m` and `CTRL-h`, respectively. This is what you will see if you insert `<CTRL-v>` <ENTER> and `<CTRL-v>` <BACKSPACE>.

### REPLACING, CHANGING, AND DELETING

#### Replacing

There are several commands you can use to replace information that is already in the editing buffer. These are the commands to use to make changes.

The simplest of these commands is the r command. This command allows you to replace exactly one character. (*r* stands for "replace.") Whatever character you type after the r replaces the character at the current cursor position.
For example, say that you mistype the word car as cqr. Move the cursor to the q and type ra. vi will replace the q with an a.

The r command is a quick way to replace one character without having to change to insert mode. The other commands in this section do change to insert mode.

The R command allows you to replace characters by typing over them. Position the cursor where you want to start the replacement and type R. You are now in insert mode. Whatever you type directly replaces what is on the screen.

For example, say that you want to change the line:

This is a bad line.

by replacing everything from the b to the end of the line. Move the cursor to the b and press R. You can now directly replace the characters as you wish. When you are finished, press <ESC>.

The s command is similar to the r command in that it allows you to replace exactly one character. However, the s command changes to insert mode to allow you to insert as many characters as you want. (s stands for "substitute.")

For example, say that you want to change cat to credit. Position the cursor to the a and type s. You are now in insert mode, and everything you type will replace the a. In this case, type redl and press <ESC>. The whole thing would look like:

sredl<ESC>

As soon as you type the s, vi erases the character that will be replaced and displays a $ as a marker. This is to remind you that you are in the middle of a replacement. The $ has nothing to do with the $ command.

If you want to replace an entire line, move the cursor to that line and press S. vi will erase the line and change to insert mode. Whatever you type replaces the line.

Sometimes you may want to replace a particular object: a sentence, a word, or a paragraph. To do this, type c, followed by the command to move the cursor over the characters that you want to replace. For example, to replace a word, move the cursor to the beginning of that word and type cw; to replace a sentence, use c); and so on.

vi will change to insert mode and display a $ character to mark the end of the characters that will be replaced. Type whatever you want to as a replacement, and then press <ESC>.

The c command is powerful. You can combine it with any cursor movement command that does not use the <CTRL> or cursor-control keys. Here are some examples. Make sure that you understand how they work.
c4w        replace 4 words
C4W        replace 4 words separated by space characters
C(         replace back to beginning of sentence
C4b        replace back to beginning of fourth previous word
C}         replace to beginning of paragraph

Note: You cannot use c with \texttt{CTRL-u, <CURSOR-UP>}, and so on. However, you can use \texttt{<SPACE>, <ENTER>, and h, j, k, and l.}

The command \texttt{C} will replace all the characters from the position of the cursor to the end of the line. The command \texttt{cc} will replace the entire line. (\texttt{cc} is the same as \texttt{S}.) These commands are not followed by a cursor movement.

As with other \texttt{vi} commands, you can execute the replace commands repeatedly by preceding them with a number. Here are some examples. Notice the subtle distinction between \texttt{4r} and \texttt{4s}.

\begin{itemize}
  \item \texttt{4r} replace 1 character with 4 copies of the insertion
  \item \texttt{4s} replace 4 characters
  \item \texttt{4cw} replace 4 words (same as \texttt{c4w})
  \item \texttt{4S} replace 4 lines
\end{itemize}

Here is a summary of the commands in this section. Take a moment now to make sure you understand how each one works. \texttt{move} stands for any cursor movement command that does not use the \texttt{<CTRL>} or cursor control keys.

\begin{itemize}
  \item \texttt{r} replace exactly one character
  \item \texttt{R} replace by typing over
  \item \texttt{s} replace one character by insertion
  \item \texttt{S} replace current line by insertion
  \item \texttt{cmove} replace from cursor to \texttt{move} by insertion
  \item \texttt{C} replace from cursor to end of current line by insertion
  \item \texttt{cc} same as \texttt{S}
\end{itemize}

All of these commands, except \texttt{r}, change to insert mode.

\section*{Changing the Case of Letters}

The \texttt{\textasciitilde} (tilde) command changes the case of a letter and then moves the cursor one position to the right. That is, if the letter is upper case, \texttt{\textasciitilde}
changes it to lower case; if the letter is lower case, \( \text{~} \) changes it to upper case. The reason the \( \text{~} \) moves the cursor is to make it easy to change consecutive letters.

For example, say that the current line is:

\[ \text{tHIS is the current line} \]

and the cursor is at the \( t \) in \text{tHIS}. If you type \( \text{~} \), the current line will look like:

\[ \text{THIS Is the current line} \]

and the cursor will be at the \( H \) in \text{THIS}. If you type \( \text{~} \) three more times, the current line will look like:

\[ \text{This is the current line} \]

and the cursor will be at the \text{space} after \text{This}.

Unfortunately, you cannot precede the \( \text{~} \) command with a number to change the case of a group of letters. However, since \( \text{~} \) does move the cursor to the right, you can use the command repeatedly to change consecutive letters. \( \text{~} \) will pass over characters that are not letters, without changing them. Thus, if you want to convert a long group of letters from one case to another, move the cursor to the beginning of the group and hold down the <\text{~}> key.

### Forming and Breaking Lines

When you type in insert mode, there are two ways that you can control the position of the right-hand margin. First, you can press <ENTER> whenever you want to start a new line. This is similar to the carriage return on a typewriter. Pressing <ENTER> inserts a \text{newline} into the editing buffer at that point; and \text{vi} assumes that all lines end with a \text{newline}.

If you type a lot, it is inconvenient to have to put in your own \text{newline} characters. Instead, you can have \text{vi} do it for you automatically, whenever you near the end of the screen line. To do this, you use an \text{ex} command. Type \text{:set wm=} followed by the distance from the left-hand margin of the screen that you want the automatic breaking to take place. (\text{wm} stands for "wrap margin."

For example, if you want \text{vi} to insert a \text{newline} whenever you move within ten positions of the left-hand margin,

\text{enter: } \text{:set wm=10}
If you want as long a line as possible, use:

```bash
:set wm=1
```

It is usually a good idea to not set `wm` too small, to allow some room on the lines for small changes. To turn off the automatic margin control,

```bash
:set wm=0
```

If you type without inserting a `newline` character (either yourself or automatically), you will be creating one very long line. If you type a line that is longer than the screen width, `vi` will display it over more than one screen line. However, `vi` still considers it to be one long line.

You will find it awkward to work with lines that do not fit on one screen line. You should avoid creating such lines by pressing `<ENTER>` as you insert or by setting the automatic margin control.

However, in the course of your work, you will often create extra long lines when you make changes. At such times you will need a way to break a line into two.

To do this, all you have to do is insert a `newline` character where you want to break the line. For example, say that the current line is:

```
This line is much too long and should be broken into two lines.
```

You decide to break the line in between the `and` and `should`.

Move the cursor to the `space` between the `d` of `and` and the `s` of `should`. Then, type `r<ENTER>`. This replaces the `space` with a `newline`, effectively breaking the lines. It will now look like this:

```
This line is much too long and should be broken into two lines.
```

Sometimes, you will want to perform the opposite operation. You will want to join two short lines into one long line. This is even easier. Move the cursor to the first line and type `J`. (`J` stands for "join.") `vi` will automatically delete the `newline` at the end of the first line, effectively joining it to the second line. If you want to join a number of lines, type that number before the `J`. For example, to join five lines, use `5J`.

When `vi` joins lines, it automatically puts in `space` characters where appropriate. For example, `vi` will put a `space` between the last word of the first line and the first word of the second line. If the end of the first line was the end of a sentence, `vi` will insert two `space` characters.

If you are typing large documents, you will find it time-consuming to line up the margins and to readjust them every time you make changes. In this case, you should use one of the text formatting programs that comes with Xenix (`nroff` and `troff`) to process your documents.
Deleting

There are four commands that you can use to delete information from the editing buffer. The \texttt{x} command deletes one character at the position of the cursor. For example, say that the current line is:

\begin{verbatim}
This is the current line
\end{verbatim}

and you want to delete the \texttt{R} in \texttt{current}. Move the cursor to the \texttt{R} and type \texttt{x}.

You can type a number before \texttt{x} to delete that many characters, starting from the cursor and moving right. For example, to delete five characters, type \texttt{5x}. Another way to delete consecutive characters is to move the cursor to the first character and either press \texttt{<x>} repeatedly or hold down the \texttt{<x>} key. (Think of typing \texttt{x} characters on a typewriter to cross out a mistake.)

The \texttt{X} command deletes one character to the left of the position of the cursor. For example, say that the current line is:

\begin{verbatim}
abcdefg
\end{verbatim}

and the cursor is at the \texttt{d}. If you type \texttt{x}, \texttt{vi} will delete the \texttt{d}. If you type \texttt{X}, \texttt{vi} will delete the \texttt{c}.

You can type a number before \texttt{X} to delete that many characters, moving left from the position of the cursor. For example, if the current line is:

\begin{verbatim}
abcdefg
\end{verbatim}

and the cursor is at the \texttt{d}, \texttt{3X} will delete \texttt{abc}.

There is a more powerful delete command, the \texttt{d} command. The \texttt{d} command is analogous to the change command. You type \texttt{d}, followed by a cursor movement command, and \texttt{vi} deletes from the cursor to where the cursor would be after being moved.

For example, to delete from the cursor position to the end of a word, type \texttt{dw}; to delete five words, type \texttt{d5w}; to delete backwards to the beginning of the third previous sentence, type \texttt{d3(}; and so on. As with the \texttt{c} command, you can use any cursor movement command that does
not use the <CTRL> or cursor control keys. A particularly useful combination is to type \texttt{dG} to delete from the cursor to the end of the editing buffer.

To delete the current line, type \texttt{dd}. To delete from the position of the cursor to the end of the current line, type \texttt{D}. (These two commands are analogous to the \texttt{cc} and \texttt{C} commands.)

Here is a summary of the commands in this section. Take a moment now to make sure you understand how each one works. \textit{move} stands for a cursor movement command that does not use the <CTRL> or cursor-control keys.

\begin{itemize}
  \item \texttt{x} \hspace{1cm} \text{delete character at position of cursor}
  \item \texttt{X} \hspace{1cm} \text{delete character to left of cursor}
  \item \texttt{dmove} \hspace{1cm} \text{delete from cursor to move}
  \item \texttt{dd} \hspace{1cm} \text{delete current line}
  \item \texttt{D} \hspace{1cm} \text{delete from position of cursor to end of line}
\end{itemize}

Remember, to undo any of these commands, use the \texttt{u} command.

**USING THE BUFFERS**

**Recovering Deletions Using the Delete Buffers**

\textit{vi} maintains nine buffers called DELETE BUFFERS. These buffers are designated by the numbers \texttt{1} through \texttt{9}. The delete buffers hold the last nine deletions that involved a line, sentence, or anything longer. (In other words, these buffers are not updated when you delete words or characters.)

Buffer \texttt{1} holds the most recent such deletion, \texttt{2} holds the next most recent, and so on. At any time, you can recover these deletions by using the \texttt{p} and \texttt{P} commands. (\texttt{p} stands for "put."

To recover the information in a delete buffer, type " (double-quote), followed by the number of the delete buffer you want, followed by \texttt{p} or \texttt{P}. (Notice that this is the double-quote character, not two single-quote characters.) For example, to recover the third last deletion, type "\texttt{3P}.

The \texttt{p} and \texttt{P} commands both insert the contents of the specified delete buffer. The only difference is that \texttt{p} places the insertion to the right of the cursor, and \texttt{P} places the insertion to the left of the cursor. If the insertion is a full line or more, \texttt{p} opens a new line above the current line, while \texttt{P} opens a new line below the current line.

For example, if you delete an important line, you can get back by using the \texttt{u} command. However, \texttt{u} only undoes the last change to the editing buffer. What if you do not realize that you want the line back
Using the Buffers

until after you have deleted another line? In this case, it is too late to use the \texttt{u} command. Since you have done a second deletion, the original one is in delete buffer 2. Move the cursor to where you want to insert and type "2P."

You can use the . (period) command to repeatedly execute the \texttt{p} and \texttt{P} commands. Usually, . repeats the last command that changed the editing buffer. However, as a special case, when you use . to repeat a command that uses a delete buffer, . automatically increases the number of the buffer by 1.

For example, say you want to recover a deletion, but you are not sure which buffer is the right one. Move the cursor to where you want to insert and type "1P. If it was not the right buffer, type \texttt{u} to undo the change. Then type . to repeat the command. You will see the contents of buffer 2, just as if you had typed "2P. If this is not the right buffer, type \texttt{u} again. Keep going until you get the buffer you want.

Remember this pattern; it will come in handy:

"1Pu.u.u.u.

The contents of the delete buffers are lost when you stop \texttt{vi}, but not when you start editing another file. However, it is not a good idea to use the delete buffers to pass information between files. Use the named buffers (see next section).

Moving Information Using the Named Buffers

\texttt{vi} has a set of buffers that you can use as temporary storage areas. These buffers are designated by the letters of the alphabet \texttt{a} through \texttt{z} and are called the NAMED BUFFERS. \texttt{vi} has a number of commands to allow you to move information back and forth between a named buffer and the editing buffer. You can copy information into a named buffer and then insert the information anywhere in the editing buffer.

To copy information from the editing buffer into a named buffer, use the \texttt{y}, \texttt{Y}, and \texttt{yy} commands (\texttt{y} stands for "yank.") These commands do not modify the editing buffer, they merely copy from it.

Using the \texttt{y} command is similar to using the \texttt{c} and \texttt{d} commands. You follow the \texttt{y} with a cursor movement command. \texttt{vi} copies from the current position of the cursor up to the position to which the command would move the cursor. You precede \texttt{y} with " (double-quote) and the name of the buffer to which you want to copy.

For example, to copy five words to the \texttt{a} buffer, type "ay5w; to copy four sentences to buffer \texttt{b}, type "by4); to copy ten characters to buffer \texttt{c}, type "cy10<SPACE>".
Creating and Modifying Text Files

(As with the c and d commands, you cannot use a cursor movement command that involves a cursor-control key or the <CTRL> key.)

The Y and yy commands both copy an entire line to the buffer you specify. For example, to copy the current line to buffer c, type either "cY or "cyy.

(Note: There is a subtle inconsistency here. The y, c, and d commands behave similarly, as do the yy, cc, and dd commands. However, the Y command copies the entire current line, while the C and D commands only process from the position of the cursor to the end of the line. Thus, Y is the same as yy, while C and D are not the same as cc and dd.)

You can repeat the y, Y, and yy commands by typing a number in front of them. For example, to copy ten lines to buffer d, type "d10yy.

Once you have copied information into a named buffer, you can insert it anywhere you want in the editing buffer by using the p and P commands. Use these commands in the same way as when you are working with the delete buffers. (See the previous section.) For example, to copy information from the buffer e to the editing buffer, use "ep or "eP. The p and P commands do not change the name buffer; they merely copy from it. Thus, you can copy from the same named buffer over and over.

There is one more way to put information into a named buffer. If you precede a delete command with a double-quote and the name of a buffer, whatever is deleted will be copied to the buffer. This works with all three delete commands: d, dd, and D.

For example, to delete ten words, you would type d10w. If you type "fd10w the words are deleted, but they are saved in buffer f. This would be the same as fy10w followed by d10w.

When you copy information to a named buffer, the information replaces whatever was in the buffer. Sometimes, it is convenient to add on to the end of a buffer, rather than replace the contents. To do this, simply specify the buffer name as an upper-case letter, rather than as a lower-case letter.

For example, "gy10w replaces the contents of buffer g with the ten words that are copied; "Gy10w adds the ten words on to the end of buffer g.

Here is how this might come in handy: Suppose you are writing an instruction manual and you want certain lines to be repeated at the end, as part of the summary. Go through the editing buffer and find the lines. For the first one, copy it to a named buffer, say h, by typing "hyy. For each subsequent line, add it on to the end of the buffer by using "Hy. Once you have gathered all the lines, move the cursor to where they are to be inserted and type "hp.

The information in the named buffers is lost when you stop vl, but
not when you start editing another file. Thus, an important use of the named buffers is to pass information between files. Simply put the information into a named buffer and switch files. The named buffer is unchanged, and you can insert the information into the new editing buffer.

Recovering the Last Deletion Using the Unnamed Buffer

There is one more buffer that you need to know about. It does not have a particular name, so it is called the UNNAMED BUFFER.

As you know, vi uses the delete buffers to save your nine most recent deletions. However, these are only deletions that are at least as long as a sentence or a line. vi uses the unnamed buffer to save your most recent deletion, no matter what size it was.

For example, if you type 10dd to delete ten lines, vi will save the lines in both the unnamed buffer and in delete buffer 1. But if you type 10x to delete ten characters, vi will save the characters only in the unnamed buffer; delete buffer 1 will remain unchanged.

If you use the p and P commands without specifying a specific buffer, they will use the unnamed buffer. Do not use a double-quote character, simply type p or P.

For example, say that you want to move a sentence from one paragraph to another. Move the cursor to the beginning of the sentence, and type d) to delete the sentence. Then move the cursor to where you want the sentence and type P.

Remember, vi will replace the contents of the unnamed buffer the next time you delete anything; if you want to save something, put it into a named buffer.

Here are two ways to use the p command with the unnamed buffer that are so handy you should memorize them outright. To swap two adjacent characters, type xp, and to swap two consecutive lines, type ddp.

For example, say the current line is:

This is the current line

and you want to change line to lnie. Move the cursor to the n in lnie and type xp. The x command deletes the n character and puts it into the unnamed buffer. The p command copies the n character from the unnamed buffer back to the editing buffer. However, after the deletion, the cursor is at the i, so the n is inserted to the right of the i. The net effect is to swap the two characters.

Typing ddp works analogously. The dd command deletes the current line, and the p command copies it back to the editing buffer
below the new current line. The net effect is to swap the two lines.

The contents of the unnamed buffer are lost when you stop vi but not when you start editing another file. However, if you want to move information between files, it is better to use a named buffer.

You might wonder, Is the unnamed buffer the same place that vi keeps the last deletion for use by the u command? The answer is no: The u command undoes the last deletion or insertion; the unnamed buffer holds only the last deletion. The unnamed buffer is not changed when you do an insertion.

Here is a summary of the commands to use the buffers. Take a moment to make sure you understand how each one works. move stands for any cursor movement command that does not use a <CTRL> or cursor-control key. buf stands for the name of any delete buffer (1 through 9) or any named buffer (a through z). BUF stands for the uppercase name of any named buffer (A through Z).

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>copy from unnamed buffer; insert after cursor</td>
</tr>
<tr>
<td>P</td>
<td>copy from unnamed buffer; insert before cursor</td>
</tr>
<tr>
<td>&quot;bufp&quot;</td>
<td>same as p but use specified buffer</td>
</tr>
<tr>
<td>&quot;bufP&quot;</td>
<td>same as P but use specified buffer</td>
</tr>
<tr>
<td>ymove</td>
<td>copy to unnamed buffer from cursor position to move</td>
</tr>
<tr>
<td>yy</td>
<td>copy one line to unnamed buffer</td>
</tr>
<tr>
<td>Y</td>
<td>same as yy</td>
</tr>
<tr>
<td>&quot;bufymove&quot;</td>
<td>same as y but use specified buffer</td>
</tr>
<tr>
<td>&quot;bufyy&quot;</td>
<td>same as yy but use specified buffer</td>
</tr>
<tr>
<td>&quot;bufY&quot;</td>
<td>same as Y but use specified buffer</td>
</tr>
<tr>
<td>&quot;BUFymove&quot;</td>
<td>same as y but add onto specified buffer</td>
</tr>
<tr>
<td>&quot;BUFyy&quot;</td>
<td>same as yy but add onto specified buffer</td>
</tr>
<tr>
<td>&quot;BUFY&quot;</td>
<td>same as Y but add onto specified buffer</td>
</tr>
<tr>
<td>&quot;bufdmove&quot;</td>
<td>same as d but save in specified buffer</td>
</tr>
<tr>
<td>&quot;bufdd&quot;</td>
<td>same as dd but save in specified buffer</td>
</tr>
<tr>
<td>&quot;bufD&quot;</td>
<td>same as D but save in specified buffer</td>
</tr>
<tr>
<td>&quot;BUFdmove&quot;</td>
<td>same as d but save onto end of specified buffer</td>
</tr>
<tr>
<td>&quot;BUFdd&quot;</td>
<td>same as dd but save onto end of specified buffer</td>
</tr>
<tr>
<td>&quot;BUFD&quot;</td>
<td>same as D but save onto end of specified buffer</td>
</tr>
</tbody>
</table>
INTRODUCTION

SETTING UP YOUR vi WORKING ENVIRONMENT

ACCESSING THE SHELL

IMPORTANT ex COMMANDS

WORKING WITH LINE RANGES AND PATTERNS

SOME USEFUL SERVICES

USING vi TO EDIT PROGRAMS
INTRODUCTION

This is the last of the chapters describing the vi editor. This chapter will teach you about using the shell from vi, important ex commands, and setting up your working environment. For programmers, there are a few sections describing special features to help you edit programs.

New Words in This Chapter

OPTION
a value you can set that lets you specify how you want vi to behave

DEFAULT
for each option, the value that is automatically assigned when vi starts

SWITCH OPTION
an option that has a value of either on or off

STRING OPTION
an option that has the value of a string of letters or numbers

EDITING SCRIPT
a text file that contains ex commands

TAG FILE
a text file that contains information about particular locations in a group of files.

SETTING UP YOUR vi WORKING ENVIRONMENT

Using vi with Different Terminals

vi was designed to take full advantage of the display screen. To do so, vi must know the characteristics of your terminal. If you are using the console, this is automatic. However, if you are using a remote terminal, you must tell vi what type of terminal it is.

Xenix maintains a file that holds descriptions of many different types of terminals. The file is called /etc/termcap. (termcap stands for "terminal capabilities." ) For vi to be able to work with a terminal, its
description must be in this file.

Each terminal in the /etc/termcap file has a name that Xenix uses to refer to that terminal. It is this name that you must tell vi. To find the name for your terminal, look in the Xenix Command Reference Manual, in the "TERMINALS" entry in Section 2. This entry lists some but not all of the terminals in /etc/termcap.

If your terminal is not listed, you will have to look in /etc/termcap. Use the more program to display the file. Search for a lower-case string that is the name of your terminal. (For example, if you have an ADM terminal, you would search for adm.) When you find the entry for your terminal, use the first name on that line. If you have problems, ask the system manager for help.

Once you know the Xenix name for your terminal, you must tell it to vi. How you do so depends on which shell you are using. If you are using the Bourne shell, you add two lines to your .profile file. If you are using the C-Shell, you add one line to your .login file. If you are using the visual shell, the terminal name has already been set.

(Each shell has a file that it executes automatically whenever you log in. The file must be in your home directory. The Bourne shell executes a file called .profile, and the C-Shell executes a file called .login. This is discussed in Chapter 21.)

If you are using the Bourne shell, add the following lines to your .profile file:

```bash
TERM=name
export TERM
```

where name is the name of your terminal. If you are using the C-Shell add the following line to your .login file:

```bash
setenv TERM name
```

where name is the name of your terminal.

Once this is set up, vi will automatically know what type of terminal you have, and there shouldn’t be any problems.

---

### Setting and Displaying Options

An OPTION is a value you can set that lets you specify how you want vi to behave. Each option controls one facet of vi’s behavior. When you start vi, it automatically assigns each option a value, called a DEFAULT. If you want, you can change the value of any option to be different from the default.
There are two types of options: SWITCH OPTIONS and STRING OPTIONS. A switch option is either on or off, like a light switch. When you start `vi`, some of the switch options are set on as a default, while others are set off.

An example of a switch option is the `number` option that was described in Chapter 11. When the `number` option is on, `vi` displays a line number in front of each line. When the number option is off, `vi` does not display a line number.

A string option has the value of a string of letters or numbers. An example of a string option is the `wm` option described in Chapter 12. The value of `wm` is a string, consisting of a number, that specifies the position for automatic margin control.

To display the current value of an option or to set an option, you use the `ex:set` command. To display a list of all the options and their current values,

```plaintext
enter: :set all
```
To display a list of those options whose values have been changed from their default,

```plaintext
enter: :set
```
To display the current value of a particular option,

```plaintext
enter: :set name ?
```
where `name` is the name of the option. For example, to display the current value of the `wm` option,

```plaintext
enter: :set wm ?
```
To turn on a switch option,

```plaintext
enter: :set name
```
where `name` is the name of the switch option. To turn off a switch option,

```plaintext
enter: :set noname
```
For example, to turn on the `number` option,

```plaintext
enter: :set number
```
To turn off the `number` option,

```plaintext
enter: :set nonumber
```
Notice that you do not put a `space` between the `no` and the name of the option.
To set a string option to a particular value, enter:

```
:set name=value
```

where name is the name of the string option and value is the value that you want to give it. For example, to set the `wm` to the value `10`, enter:

```
:set wm=10
```

Notice that you do not put a `space` on either side of the `=` character.

**The Different Options**

`vi` has many different options. Here is a description of the important ones. For a full list, see the Xenix Command Reference Manual, under the section describing the `vi` command.

Some options have a full name and an abbreviation. `vi` displays the full name, but when you set the option, you can refer to it by the abbreviation.

**autowrite**

- **abbreviation:** `aw`
- **type:** switch
- **default:** off
- **effect:** automatically write the contents of the editing buffer to the current file, whenever you use a command that might change the current file.

**ignorecase**

- **abbreviation:** `ic`
- **type:** switch
- **default:** off
- **effect:** do not distinguish between upper and lower case when matching regular expressions.

**list**

- **type:** switch
- **default:** off
- **effect:** display invisible characters; `tab` is displayed as `^I` (upper-case letter "i"); at the end of each line `newline` is marked by `$`.

**mesg**

- **type:** switch
- **default:** on
- **effect:** turning off this switch prevents messages from being sent to your terminal while you are using `vi`; this is like taking your phone off the hook.
number
abbreviation: nu
type: switch
default: off
effect: display each line with its line number; this is a useful option to keep on all the time

report
type: string
default: 5
effect: display a message whenever a command modifies more than the specified number of lines

term
type: string
default: taken from the TERM variable
effect: the name of the type of terminal being used

warn
type: switch
default: on
effect: if the editing buffer has not been saved, display a warning message every time a shell command is executed

window
type: string
default: varies
effect: use this number of lines for the screen display; the default varies with the type of terminal:
- console of the IBM PC AT: full screen
- fast terminal (over 1200 baud): full screen
- medium terminal (1200 baud): 16 lines
- slow terminal (300 baud): 8 lines

wrapmargin
abbreviation: wm
type: string
default: 0
effect: during insert mode, automatically break a line into two (by inserting a newline), this many characters from the right end of the line; to turn off this option, specify a value of zero (the default)
**ACCESSING THE SHELL**

**Using XENIX Commands from vi**

There are several `ex` and `vi` commands that you can use to execute Xenix commands from `vi`. This saves you from having to stop `vi` every time you want to use a Xenix command.

To execute one Xenix command,

```
enter: :!command
```

where `command` is a Xenix command. For example, to find out the time without stopping `vi`,

```
enter: :!date
```

To repeat the last such command,

```
enter: :!l
```

If you want to execute a number of Xenix commands, you can pause `vi` and start a new copy of the shell. To do this,

```
enter: :ish
```
You can now enter as many Xenix commands as you want. When you are finished, press <CTRL-d> (the eof signal). Xenix will stop the shell and restart vi.

You can execute a Xenix command and insert the output into the editing buffer by using a form of the :re command. (re stands for “read”—the :re command is explained more fully later in this chapter.)

```
enter: :linere !command
```

where line is the line number at which you want the insertion to take place, and command is a Xenix command. If you do not specify a line number, vi will insert at the current line.

For example, to insert the output of the date command into the editing buffer at the current line, use:

```
:re ldate
```

So far, all the commands in this section have been ex commands. There are two vi commands that you can use to have a Xenix command act on part of the editing buffer. vi will replace the part of the editing buffer that was processed with the output of the command. For example, you can insert a table of information and then have Xenix sort it for you.

The first way to use this facility is to enter:

```
!move command
```

where move is a cursor movement command and command is a Xenix command. As with the c, d, and y commands, the cursor movement command cannot use the <CTRL> or cursor-control keys.

vi will execute the Xenix command on part of the editing buffer; starting from the position of the cursor, up to where the cursor movement command would move the cursor.

For example, say that you want to count the number of characters in a group of four sentences and insert that number into the editing buffer. First, make a copy of the sentences and position the cursor at the beginning of the first sentence. Next,

```
enter: !4)wc -w
```

The output of the wc -w command (the number of characters) will replace the sentences. (This is why you execute the Xenix command on a copy of the sentences.)

After you type the cursor movement command (in this case 4), vi will display a ! on the command line. As you type the actual Xenix command, vi will echo it on the command line, after the ! character.
This is the same as when you enter an ex command or a search (/, ?) command. As with these types of commands, you can correct what is being echoed by using <BACKSPACE>, <CTRL-w>, and <CTRL-u>. To cancel the command before you press <ENTER>, use <DEL>. If you are not satisfied with the results of the command, you can undo it by typing u.

There is an alternate way to have a Xenix command act on part of the editing buffer, if it consists of a number of lines.

```
enter: n!!command
```

where n is the number of lines, starting with the current line; and command is the Xenix command. (After the second !, vi will display a ! on the command line and echo the Xenix command.)

For example, say that you want to sort ten lines in alphabetical order. You can use the sort program (discussed in Chapter 16) to replace the lines by sorted lines. First, move the cursor to the first line. Next,

```
enter: 10!!sort
```

(In this case, you don't need to make a copy of the lines because you want them to be replaced.) This last command is equivalent to:

```
!10|sort
```

(Remember, 10| will move the cursor down ten lines.)

Here is a summary of the commands in this section. Take a moment to make sure that you understand each one. command stands for a Xenix command. line stands for a line number. move stands for any cursor movement command that does not use the <CTRL> or cursor-control keys. n stands for a number.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>:lcommand</td>
<td>execute the specified Xenix command</td>
</tr>
<tr>
<td>:!!</td>
<td>repeat previous Xenix command</td>
</tr>
<tr>
<td>:!sh</td>
<td>pause vi and start a new copy of the shell</td>
</tr>
<tr>
<td>:linere! command</td>
<td>insert Xenix command output at specified line</td>
</tr>
<tr>
<td>:re! command</td>
<td>insert Xenix command output at current line</td>
</tr>
<tr>
<td>lmove command</td>
<td>execute Xenix command from cursor to move</td>
</tr>
<tr>
<td>n!!command</td>
<td>execute Xenix command on n lines</td>
</tr>
</tbody>
</table>
IMPORTANT ex COMMANDS

Specifying Line Numbers for ex Commands

You have already learned a number of useful ex commands. There are a few more that you should know because they perform tasks that cannot be done easily by vi commands. Some of these ex commands require that you specify one or two line numbers. This section explains how to do so.

When you use an ex command, you put the line numbers on which the command is to act before the name of the command. For example, to use the :d command on line 10, you would enter:

:10d

(The :d command, which deletes whole lines, is explained later in this chapter.)

When you want a command to act on a range of numbers, you specify the first and last number of the range, separated by a comma. For example, to use the :d command on lines 10 through 15 inclusive, you enter:

:10,15d

The most straightforward way to specify a line number is to use an actual number, as in the two examples above. The question is, How do you know the numbers of the lines? There are two ways.

First, if you set the number option to on, vi will display a line number in front of each line. Many people prefer to keep this option on all the time.

Second, to find out the line number of the current line only, enter :nu. (nu stands for “number.”) vi will display the current line along with its number.

There are some ex commands that can act on one line. If you use one of these commands without specifying a line, the command will use the current line.

For example, to use the :d command on the current line,

enter:  :d

There are a number of abbreviations that you can use to specify line numbers:

. (period) the current line
$ the last line in the editing buffer
% the entire editing buffer
Here are some examples that use the :d command on different ranges of lines. To the right of each example, I have indicated which lines are specified:

- `:d` current line
- `:.d` current line
- `:$d` last line
- `:%d` entire editing buffer
- `:1,$d` line 1 through last line (same as %)
- `:1,.d` line 1 through current line
- `:.,$d` current line through last line

You can also specify a line by using one of the commands. `mark` stands for the name of a marker set by an `ma` command.

- `/` search forward for a pattern
- `?` search backward for a pattern
- `''` go to position of previous absolute move command
- `'mark` go to marker

(These commands are explained in Chapter 11. The last two are two single-quotes and one single-quote, respectively.)

Each one of these commands can take the place of a line number. If you use a `/` or `?` command, you must use a second `/` or `?` at the end of the pattern.

For example, to have the :d command act on the next line that contains the string `hello`,

```
enter:  :/hello/d
```

To search backward for the string `begin`, and search forward for the string `end`, and have the :d command act on this range of lines,

```
enter:  :?begin?,/end/d
```

The `'` and `''` commands are often useful to specify a range of lines. For example, say that you use the command 100G to go to line 100. You then decide to have the :d command act on the range of lines between 100 (the current line) and the line you came from.

```
enter:  ..,'''d
```

As another example, say that you have used the `m` command to mark a line with the marker `a`. Later, you want to use the :d command on all the lines from line 50 to the marker `a`.

```
enter:  :50,'ad
```
There is one more facility for specifying line numbers. If \( n \) represents a number, you can refer to the line that is \( n \) lines before a particular line by using \(-n\). Similarly, \(+n\) means after a particular line.

For example, to use the \( :d \) on the line following the next occurrence of the string \texttt{hello},

\[
\text{enter: } :/\texttt{hello}/+1d
\]

Here is an example that uses \( :d \) on a range of lines: The range starts two lines before the last occurrence of the string \texttt{begin} and ends three lines after the next occurrence of the string \texttt{end}.

\[
:?\texttt{begln}\neg2,/\texttt{end}/+3d
\]

If you use \(-n\) and \(+n\) by themselves, they refer to before and after the current line, respectively. For example, here are two equivalent ways of using the \( :d \) command, starting from four lines before the current line, up to five lines after the current line (a total of ten lines).

\[
:-4,+5d
\]

\[
:-4,+5d
\]

Here is a summary of the ways to specify line numbers for \texttt{ex} commands. Take a moment and make sure that you understand how each one works. \texttt{pattern} stands for a regular expression describing a pattern. \texttt{mark} stands for the name of a marker set by an \texttt{m} command.

You can specify one line number or two line numbers separated by a comma to indicate a range.

- current line
- last line
- entire file (same as \texttt{1,$})
- search forward: next line containing \texttt{pattern}
- search backward: next line containing \texttt{pattern}
- line of previous absolute move command
- line containing \texttt{mark}

To indicate a line before or after a specific line:

\[
:-n \quad \text{\( n \) lines before}
\]

\[
:+n \quad \text{\( n \) lines after}
\]

If you do not specify a line, most commands use the current line.
Writing and Reading Using ex

In Chapter 10, I explained how to use the :w and :wl commands to write from the editing buffer to a file. By themselves, these commands copy the whole editing buffer to a file. If you want, you can copy a line or a range of lines. Simply specify the lines before the w.

For example, to copy lines 50 to 120 to the file temp,

```
enter: :50,120w temp
```

When you use the :w or :wl commands, whatever you write replaces the file to which it is being written. If you want to add to a file, rather than replace it, use :w>>. vi will copy the information to the end of the file.

For example, to copy lines 50 to 120 to the end of the file temp,

```
enter: :50,120w>> temp
```

You can copy information from a file into the editing buffer by using the :re command. (:re stands for "read.")

```
enter: :linere file
```

where line stands for a line number and file stands for the name of a file.

For example, to copy the contents of the file temp to the end of the editing buffer,

```
enter: :$re temp
```

An alternate form of the read command lets you copy the output of a Xenix command into the editing buffer.

```
enter: :linere! command
```

where line is a line number and command is a Xenix command.

For example, to insert after line 150 a list of the files in your working directory,

```
enter: :150re! lc
```

Here is a summary of the commands in this section. Take a moment to make sure that you understand each one. line stands for a line number. file stands for the the name of a file. command stands for a Xenix command. If you don't specify a line number, vi will use the current line.
Deleting, Copying, and Moving Using ex

You can use ex commands to delete, copy, and move groups of lines. Although it is possible to do this with vi commands, the ex commands are more convenient when you are working with whole lines.

To delete one or more lines, use the :d command. vi will delete whatever lines you specify. For example, to delete line 100,

```
enter: :100d
```

To delete from the current line to the next line containing the string hello,

```
enter: :./hello/d
```

To delete the entire editing buffer, use:

```
:%d
```

To copy one or more lines, use the :co command. (co stands for “copy.”) It looks like this:

```
:lineco target
:line, lineco target
```

where line stands for a line number, and target is the number of the line after which you want to insert the copied lines.

For example, to copy lines 5 to 10 and insert them after line 25,

```
enter: :5,10co25
```

To search backward and copy the first line with the string hello to the beginning of the editing buffer,

```
enter: :?hello?co0
```

To move a line or lines, use the :m command. (m stands for “move.”) The :m command has the same form as the :co command. The difference between the two commands is that :m deletes the original lines and then inserts them somewhere else. :co does not delete the original lines.
Here are some examples. To move lines 15 to 20 and insert them after line 25,

enter:  :15,20m25

To move the last line to the beginning of the editing buffer,

enter:  :$m0

If you make a mistake, you can undo the effects of a :d, :co, or :m command with the u command.

Here is a summary of the commands in this section. Take a moment to make sure you understand each one. line stands for a line number. target stands for the line number after which the insertion is to take place.

- :linedeleted specified line
- :line,linedelete specified range
- :linecotlinecopy specified line; insert after target
- :line,linetargetcopy specified range; insert after target
- :linemtargetmove specified line; insert after target
- :line,linemtargetmove specified range; insert after target

WORKING WITH LINE RANGES AND PATTERNS

Substituting over a Range of Lines

There are two ex commands that will search for patterns and perform substitutions. The :s command will search over a specified range, and the :g command will search over the whole editing buffer. (:s stands for “substitute”; :g stands for “global.”) Both these commands search forward.

To use the :s command, you must tell it what pattern to look for and what to substitute when it finds the pattern. The pattern can be any regular expression. The command looks like this:

:s/pattern/replace/

pattern is the regular expression for which you want to search. replace is the string that is to take the place of pattern.
Here is an example. To replace the string HELLO by the string hello,
enter: :s/HELLO/hello/

Here is a more complicated example. You want to search for a word
that begins with the letter h or H and replace the entire word by xxx.
:s/\<[Hh].*\>/xxx/

In this example:
\< matches the beginning of a word
[Hh] matches either h or H
.* matches zero or more occurrences of any
character except a newline
\> matches the end of a word

If you want to delete something, you can search for it and replace it
by nothing. For example, to delete any digit,
enter: :s/[0-9]/

By itself, the :s command will search the current line. If you want to
search a different line, or a range of lines, you can specify the line
numbers in front of the command.
For example, this command replaces hello by (hello) on line 100.
:100s/hello/ (hello)/

The following command inserts five spaces at the beginnings of lines
100 through 105:
:100,105s/^/ /
(Remember, ^ matches the beginning of a line.)
In this form, the :s command will match and replace the first
occurrence of the pattern on each line you specify. Sometimes you want
to substitute all the occurrences on the line. To do this, put a g at the end
of the command. (g stands for “global.” However, a g at the end of a
command is not the same as the :g command, which is explained below.)
Here is an example. To replace every occurrence of the string hello
on the current line by the string Hello,
enter: :S/hello/Hello/g
This next command searches for all the occurrences of the string bad,
from the current line to the last line, and replaces them with the string
good.
Working with Line Ranges and Patterns

`:,.$s/bad/good/g`

The `g` ending is particularly useful when you have discovered that you have misspelled a word many times throughout the editing buffer. For example, the following command changes every occurrence of `Diane` to `Dianne`:

`%s/Diane/Dianne/g`

(Remember, `%` specifies the entire editing buffer.)

There is one other way to modify a `:s` command. If you put `c` at the end of the command, `vi` will ask for your permission before it makes each substitution. (`c` stands for “confirm.”)

For example, say that you want to search each line from 100 to 150. Many of these lines contain the string `bad`. You want to replace the first occurrence of `bad` on some of the lines with the string `good`.

Enter: `:100,150s/bad/good/c`

For each string that is matched, `vi` will display the line and show you where the string is. If you enter `y` (for “yes”), `vi` will make the replacement. If you enter anything else (say `n` for “no”), `vi` will not make the replacement. (Remember, you press <ENTER> after you make your choice.)

You can add both `g` and `c`, in any order, to the end of a `:s` command. For example, the following will replace every occurrence of the string `bad` with the string `good`, in lines 100 to 150, subject to your confirmation.

`:100,150s/bad/good/c`

Here is a summary of the `:s` command. Take a moment and make sure that you understand the different forms. `line` stands for a line number. `pattern` stands for a regular expression. `replace` stands for the string that is to take the place of `pattern`.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>:s/pattern/replace/</code></td>
<td>substitute on current line</td>
</tr>
<tr>
<td><code>:lines/pattern/replace/</code></td>
<td>substitute on specified line</td>
</tr>
<tr>
<td><code>:line,lines/pattern/replace/</code></td>
<td>substitute over specified range</td>
</tr>
</tbody>
</table>

To replace all occurrences on the specified lines, put `g` at the end of the command. To have `vi` ask you for confirmation before making the substitution, put `c` at the end of the command.
Executing Commands over a Range of Lines

The :g command lets you search a specified range of lines and select each line on which a particular pattern occurs. You can then execute an ex command on each of these lines. (g stands for "global.")

The :g command looks like this:

```
line,line:g/pattern/excom
```

where line stands for a line number; pattern stands for a regular expression; and excom stands for one or more ex commands. If you specify more than one ex command, you must separate them with a \ (vertical bar). You can put space characters between ex commands to make them easier to read. Note: You can specify any ex command except another :g command.

For example, say you want to replace each string 5 with the string five on every line from 100 to 150 that has the string number on it.

```
enter: 100,150g/number/ s/5/flve/g
```

Here is the same command, except that two substitutions are performed on each line: 5 is replaced by five, and 6 is replaced by six.

```
100,150g/number/ s/5/flve/g | s/6/slx/g
```

Most ex commands will act on the current line if you do not specify any line numbers. The g: command (like the :w and :wl commands) will act on the entire editing buffer. For example, the following command replaces the string 5 by the string five on every line in the editing buffer that contains the string number.

```
g/number/ s/5/flve/g
```

You will often find that the string you use for the :g command is the string you want to replace in a subsequent :s command. For your convenience, vi will consider that each unspecified search pattern has the value of the last pattern that was matched.

For example, say that you want to substitute each occurrence of Diane by Dianne throughout the editing buffer. Here are three ways to do it:

```
%s/Diane/Dianne/g  
g/Diane/ s/Diane/Dianne/g  
g/Diane/ s//Dianne/g
```

In the last command, the search pattern for the :s command is not specified, so vi assumes that it is the last pattern matched; in this case Diane, from the :g command.
Sometimes, you will find that you want a command to act on each line that does not contain a particular string. In this case, use the :gl command. This command works exactly like :g except that it selects those lines that do not contain the pattern you specify.

For example, to change the string bad to good everywhere in the editing buffer, except on those lines that contain the string okay,

enter: \:gl/okay/ s/bad/good/g

Here is a summary of the :g and :gl commands. Take a moment to make sure you understand the different forms. pattern stands for a regular expression. excom stands for one or more ex commands but not another :g command. If you specify more than one ex command, you must separate them with a | (vertical bar) character. You can put space characters between ex commands to make them easier to read.

- :g/pattern/excom: execute excom on lines containing pattern
- :line,line:g/pattern/excom: same as :g over specified range
- :gl/pattern/excom: execute excom on lines not containing pattern
- :line,line:gl/pattern/excom: same as :gl over specified range

Note: You can undo all the effects of a :g command with an u command.

Matching and Substituting Complicated Patterns

When you use the :s and :g commands, you can use any regular expression to specify the search pattern. I'm sure that by now you appreciate the power of regular expressions.

In Chapter 11 you learned how to use the following characters in regular expressions:

- . matches any one character except a newline
- * matches zero or more of the preceding characters
- ^ matches the beginning of a line
- $ matches the end of a line
- < matches the beginning of a word
- > matches the end of a word
- [ ] matches one of the enclosed characters
- [^ ] matches any character that is not enclosed
Inside [ and ] you can use - to indicate a range of single letters or digits. \ indicates that the following character is to be taken literally, except for \<, \>, \(, and \\).

In this section, you will learn a few more ways to match patterns and to specify replacement strings. These facilities extend the power of the :s and :g commands.

When you specify the replacement string in a :s command, only two characters—^ (tilde) and & (ampersand)—have special meanings. All the other characters are taken literally. For example, consider the following command:

:s/^/^

The first ^ is in the search pattern, so it matches the beginning of the line. The second ^ is in the replacement string, so it literally means a ^. The effect of this command is to insert a ^ character at the beginning of the current line.

Within a replacement string, ^ and & have special meanings. ^ stands for the replacement from the previous :s command. (In fact, it also has this meaning in a regular expression.) & stands for the characters that were matched by the :s command.

Here are some examples. Say that you are typing a long letter in which the names Popeye and Popeye the Sailor are used frequently. To make the job easier, you can type P each time you want Popeye and S each time you want Popeye the Sailor. When you are finished, you can use :s commands to replace all the words consisting of P or S.

The following command replaces all the P words. Notice that we have to use \< and \> to make sure that we only replace whole words. Otherwise, we would accidentally replace all the P characters that were part of a word, such as Psmith.

:%s/\<P\>/Popeye/g

(Remember, % means the entire editing buffer.)

When we make the next substitution, we can use ^ to stand for the last replacement string—in this case Popeye.

:%s/\<S\>/~ the Sailor/g

Here is an example showing how to use & to stand for the pattern that was matched by the :s command. Say that you want to put parentheses around all the words that start with an upper-case letter. To match such a word, we would use:

\<[A-Z].*>
In other words, the beginning of a word, followed by any letter from A to Z, followed by zero or more occurrences of any letter, followed by the end of a word.

Here is a command to put parentheses around each such word in the editing buffer:

```
:s/\<[A-Z].*/1(&)/
```

See how easy it is when & stands for the pattern that has just been matched.

Sometimes you want to refer to only part of the pattern that has been matched. In that case, you can divide the pattern into parts when you specify it. In the replacement string, you can refer to the first part as \1, the second part as \2, and so on. To divide a search pattern into parts, enclose each part in \( and \).

This can get a little complicated so here are a few examples. Say that you want to search for all the strings that consist of one upper-case letter followed by one digit. (For example, A5 or B8.) You want to reverse the two characters.

The way to do it is to divide the search string into two parts. Part 1 will be the letter; part 2 will be the number. You can then replace the string with part 2 followed by part 1.

```
%s/\([A-Z]\)\([0-9]\)/\2\1/g
```

This sort of thing can become very complicated. Here is an example of a command that would give most people nightmares: Read it slowly.

We want to search for any words that contain a hyphen and remove the hyphen. (Remember, - has a special meaning in regular expression, so we must quote it when we use it literally.)

First, we cannot use a simple command such as:

```
:%s/-//g
```

to replace all hyphens by nothing because we only want to replace hyphens that are inside of words. For example, we do not want to replace - characters that may be in an arithmetic expression.

Let us assume that a word consists of an upper- or lower-case letter, followed by lower-case letters. We are interested in those words that have a hyphen in them. This can be described as:

```
\<[A-Za-z][a-z]*/-[a-z][a-z]\>
```

In other words, the beginning of a word, followed by an upper- or lower-case letter, followed by zero or more lower-case letters, followed by a hyphen, followed by a lower-case letter, followed by zero or more
lower-case letters, followed by the end of the word. (One thing about regular expressions: They may seem complicated, but they are a lot better for describing patterns than is English.)

To use this to form an appropriate :s command, all we have to do is divide it into parts. Part 1 will be the letters before the hyphen: part 2 will be the letters after the hyphen. The hyphen itself will not be inside a part. All we have to do is replace part 1, followed by a hyphen, followed by part 2, with part 1 followed by part 2. Here is the command that does it all:

:%s/\(<[A-Za-z][a-z]*\)\-\([a-z][a-z]*\)\>/1\2/g

I admit that it looks awful, but it works. Most of the time you will not need commands like this, but if someone gives you a file with 10,000 words with hyphens in the wrong places, you will be glad that vi can do this sort of thing.

**SOME USEFUL SERVICES**

**Changing the Case of Strings**

You can use the :s command to convert strings to upper and lower case. This is done by using the characters \u, \l, \U, \L, and \E in the replacement string.

The characters \u and \l cause the next character in the replacement string to be converted to upper or lower case, respectively, if that character is a letter. For example, to convert the first character of every word that begins with a, b, or c to upper case,

```
enter:  :%s/\<[abc]\u&/g
```

\< matches the beginning of a word. [abc] matches a or b or c. & stands for the pattern that was matched.

The characters \U and \L cause all subsequent letters to be converted to upper and lower case, respectively. This continues until either the end of the string is reached or \E is encountered. (E stands for “end.”)

Here is an example that converts the first word of every line to upper case. A word can be matched by:

```
\<.*\>
```

That is, the beginning of a word, followed by zero or more characters that are not a newline, followed by the end of a word. The command is:
\%s/\<.*\>/\U&/

The \U means that all the following characters, in this case & (the string that was matched), are to be converted to upper case.

Here is an example that uses \E. This command searches for all the strings Xenixsystem or xenixsystem and changes them to XENIXsystem.

The search string will be divided into two parts: Part 1 will be the pattern [Xx]enix and part 2 will be system. The conversion to upper case will be turned on for part 1 and turned off for part 2.

:\%s/([^Xx]enix\)(system)\U1\E2/g

Using an Editing Script

As you can see, it is not hard to come up with complex ex commands that perform specialized tasks. Occasionally, you will want to perform a series of such commands repeatedly. In such cases, it is frustrating and time consuming to enter all the commands over and over.

Instead, you can create a file that contains these commands. Whenever you want, you can have vi read the file and execute the commands just as if you had entered them from the terminal. Such a file is called an EDITING SCRIPT because it gives directions, just like a movie script.

Use vi to create a file that will be an editing script. Put in each command, just as you want it to execute, and then save the file. When you want to execute the editing script, use the :so command. (so stands for "source.") Enter :so, followed by the name of the file.

For example, to execute the commands in the file change, use:

:so change

Here is an example of an editing script. Say that you have several files in which you want to change the strings 0 through 9, to the strings zero through nine, respectively. Create a file with the lines:

:\%s/0/zero/g
:\%s/1/one/g
:\%s/2/two/g
:\%s/3/three/g
:\%s/4/four/g
:\%s/5/five/g
:\%s/6/six/g
:\%s/7/seven/g
:\%s/8/eight/g
:\%s/9/nine/g
If you save this file under the name `number`, you can execute the commands by entering:

```
:so number
```

### Abbreviations

There will be times when you find yourself typing a long word or phrase repeatedly. In such cases, you can use the `:ab` command to set up an abbreviation. (ab stands for “abbreviate.”) Whenever you type the abbreviation as a whole word, `vi` will automatically change it to what you want.

To set up an abbreviation, enter `:ab` followed by the short form, followed by the long form. For example, to abbreviate `Mxyzptlk` as `m`, enter:

```
:ab m Mxyzptlk
```

From now on, whenever you type `m` as a word by itself, `vi` will immediately change it to `Mxyzptlk`. The abbreviation does not have to be similar to the string it represents. For example, you could use `m` as an abbreviation for `Bizarro`.

`vi` ignores an abbreviation if you use it within a word. Only those abbreviations that are followed by a `space` or `newline` are substituted. So, if `m` is an abbreviation, `m<SPACE>` or `m<ENTER>` will be substituted; both `m` characters in `marshmallow` will be ignored.

If you ever want to type an abbreviation as a word and you don't want it substituted, type `<CTRL-v>` before the `space` or `newline`. For example, if `m` is an abbreviation and you really want an `m` by itself, type `m` followed by either `<CTRL-v><SPACE>` or `<CTRL-v><ENTER>`.

At any time, you can display a list of all the current abbreviations by entering:

```
ab:
```

Abbreviations are lost when you stop `vi` but not when you start editing another file. If you want to cancel an abbreviation, use the `:una` command. (una stands for “unabbreviate.”) Enter `:una` followed by the abbreviation that you want to cancel. For example, to cancel the abbreviation `m`, enter:

```
:una m
```

Here is a summary of the `:ab` command. Take a moment to make sure you understand how it works. short stands for the abbreviation. long stands for the string to be substituted.
Setting Up Your vi Working Environment Automatically

`vi` has a feature to help you set up your working environment automatically. Every time `vi` starts, it looks in your home directory for a file named `.exrc`. If such a file exists, `vi` will execute each line in the file as an `ex` command.

Another way to say this is that if the file `.exrc` exists in your home directory, `vi` will perform the command:

```bash
:so /usr/userid/.exrc
```

where `userid` is your userid. (Unless you change your home directory.)

The great thing about the `.exrc` file is that you can use it to set up your `vi` working environment exactly the way you want it. This is the place to put the `:set` and `:ab` commands that you use all the time. You can also put in Xenix commands by using the `:l` command. If you are using an IBM PC AT, you can set the function keys in this way.

Within the `.exrc` file, any lines that begin with a " (double-quote) character are ignored. Thus, you can put in description lines. This is especially useful if you are using complicated commands. As well, space characters at the beginning of a line are ignored, so you can indent commands to make the file easier to read.

Here is an example of a typical `.exrc` file:

```
" set the options
  :set wm=5
  :set number
  :set window=10
" set abbreviations
  :ab m Mxyzptlk
  :ab b Bizarro
" set function keys
  :lsetkey 1 'very long and difficult expression'
  :lsetkey 2 ':!date^M'
" display the date and date
  :!date
```

Notice the function key settings. `<F1>` is set to a long string. While you are editing, you can press `<F1>` instead of typing the string. This is an
alternative to using an abbreviation. \(<F2>\) is set to display the time and date. The \textbf{newline} at the end of the command is displayed as \(^{\text{^M}}\). When you are typing this line, you will have to type \(<\text{CTRL}-v><\text{ENTER}>\) instead of \(<\text{ENTER}>\).

If an \texttt{exrc} file contains a bad command, \texttt{vi} will stop executing the file at that command. Although \texttt{vi} will start, the rest of the commands in the \texttt{exrc} file will not be processed. \texttt{vi} will usually display an error message, but sometimes it goes by so fast that you miss it. Make sure that you test a new \texttt{exrc} file carefully, to make sure that all the commands executed properly.

One drawback about the \texttt{exrc} file is that it is executed every time you start \texttt{vi}. You cannot use this file to set up an environment that you use occasionally. For example, there may be some option settings and abbreviations that you use when you are editing reports, but not otherwise. You don't want to put the commands in the \texttt{exrc} file because they will be executed any time you use \texttt{vi}.

The solution is to put these commands in a separate file and access them with a \texttt{:so} command. For example, you might put a set of commands in a file called \texttt{reports}. Whenever you edit a report, you start off entering:

\begin{verbatim}
:so reports
\end{verbatim}

Although the \texttt{exrc} file must be in your home directory, the \texttt{:so} commands will look in any directory you want. For example:

\begin{verbatim}
:so /usr/harley/exfiles/reports
\end{verbatim}

**USING vi TO EDIT PROGRAMS**

**Automatic Indentation**

The remaining sections of this chapter describe the features of \texttt{vi} that are of particular interest to programmers. If you do not plan to use \texttt{vi} to write programs, you can skip the rest of the chapter.

The chief difference between editing a program and editing a document is that a program demands an indented structure. \texttt{vi} has an autoindent facility to help you edit programs. To use this facility, set the \texttt{ai} option to on. (\texttt{ai} stands for "autoindent.")

\begin{verbatim}
enter: :set ai
\end{verbatim}
Once this setting is on, `vi` keeps track of your indentation. As you type, use the `<TAB>` key to indent. Whenever you start typing a new line, `vi` will automatically insert `tab` characters to bring the indentation up to the level of the previous line.

For example, say that you are inserting a program and you have gotten this far:

```
Axxxxxxxxxxxxxxxxxxxxx
Bxxxxxxxxxxxxxxxxxxxxx
Cxxxxxxxxxxxxxxxxxxxxx
Dxxxxxxxxxxxxxxxxxxxxx
Exxxxxxxxxxxxxxxxxxxxx
```

and that you have indented by using the `<TAB>` key. When you press `<ENTER>` at the end of the `E` line, `vi` will automatically start the new line with two `tab` characters to maintain the level of indentation.

The same principle holds when you make changes. For example, if you insert a new line between the `C` and `D` lines, `vi` will automatically insert one `tab` character at the beginning of the line.

When you are typing in insert mode with the `ai` option on, `vi` will not let you use `<BACKSPACE>` to move left over an indentation. This helps you maintain the correct alignment. When you do want to move left over an indentation, press `<CTRL-d>`. `vi` will shift left one level by erasing the `tab` character.

For instance, say that you are inserting a line after the `E` line in the last example. If you want to shift left to the level of the `D` line, press `<CTRL-d>`. If you want to shift left to the level of the `B` line, press `<CTRL-d>` a second time.

In command mode, there are a few commands that you can use to change the alignment of one or more lines. This saves you from having to change to insert mode in order to insert or delete `tab` characters.

The `<` command shifts lines to the left, and the `>` command shifts lines to the right. To use these commands, type `<` or `>`, followed by a cursor movement command. `vi` will shift all the lines one level in the direction you specify; starting from the current line up to the line to which the command would move the cursor.

For example, to shift left from the current line to the bottom of the screen, type:

```
<\n
```

As with other commands of this format, the cursor movement command cannot use the `<CTRL>` or cursor-control keys. However, you can use the `<ENTER>` key. For example, to shift the current line and the next
nine lines to the right, type:

9<ENTER>

To shift the current line left or right, use << and >>, respectively. You can put a number in front of these commands to shift that number of lines.

For example, a second way to shift the current line and the next nine lines to the right is to type:

10>>

Normally, the tab and shift positions are set at every eighth column. If you want to use different values, you must change the tab and shift settings separately.

To change the tab setting, set the ts option to the size of the interval that you want. (ts stands for “tabstop.”) For example, to set the tabs at every fifth position, type:

:set ts=5

To change the shift setting, set the sw option to the size of the interval that you want. (sw stands for “shiftwidth.”) For example, to set the shift interval to every fifth position, type:

:set sw=5

The size of this setting controls the spacing of the <, >, <<, and >> commands. It also controls the shifting when you use <CTRL-d> in insert mode.

If you use the :set all command to display all the options, you will notice an option called hardtabs. This is an option you can safely ignore.

Here is a summary of the options and commands in this section. Take a moment to make sure you understand each one.

**shiftwidth**

- abbreviation: sw
- type: string
- default: 8
- effect: sets the shift interval used by the <, >, <<, and >>>; and by <CTRL-d> in insert mode

**tabstop**

- abbreviation: ts
- type: string
- default: 8
- effect: sets the interval between tab positions
The following commands can be used in command mode to align one or more lines. `move` stands for a cursor movement command that does not use the `<CTRL>` or cursor control keys. `n` stands for a number.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;move</code></td>
<td>shift left: from current line to <code>move</code></td>
</tr>
<tr>
<td><code>&gt;</code>move</td>
<td>shift right: from current line to <code>move</code></td>
</tr>
<tr>
<td><code>&lt;&lt;</code></td>
<td>shift left: current line</td>
</tr>
<tr>
<td><code>&gt;&gt;</code></td>
<td>shift right: current line</td>
</tr>
<tr>
<td><code>n&lt;&lt;</code></td>
<td>shift left: specified number of lines</td>
</tr>
<tr>
<td><code>n&gt;&gt;</code></td>
<td>shift right: specified number of lines</td>
</tr>
</tbody>
</table>

Editing C Programs

The automatic indentation features of `vi` make it especially easy to edit programs in structured languages. Nevertheless, certain languages have their own needs. This section describes the special commands and options that you can use to edit C programs.

It is often handy to be able to identify matching parentheses, braces, and square brackets. The `%` command can be used in command mode for just this purpose. Simply move the cursor to one of the following characters:

```
( ) { } < >
```

and type `%`. `vi` will move the cursor to the matching character. If there is no matching character, `vi` will beep.

There are also two commands to allow you to move from function to function. The `]]` and `[[` commands move forward and backward, respectively, to the next line that begins with a `{` character. You can use `]]` and `[[ with commands that use cursor movement commands. For example, to copy an entire function to the `a` buffer, move the cursor to the beginning of the function and type:

```
"ay]]
```

Here is a summary of these two commands:

```
[[ move cursor backward to line that begins with {
]] move cursor forward to line that begins with {
```

Using a Tag File

When you are editing C programs, it is useful to be able to jump from one function to another. To do this, you can use a file called a TAG
FILE. A tag file contains the names of various functions. For each name, the tag file records the file in which the function resides and its location in the file. In principle, a tag file can contain information about any location in any text file.

Once you set up the tag file, you can jump from one file to another by using the :ta command. (ta stands for “tag.”) Enter :ta followed by the name of the function. vi will jump to that function, even if it is in a different file. If the function is in a different file, remember to save the editing buffer before you use the :ta command.

Here is an example. To jump to the function sendmsg,

enter:  :ta sendmsg

If you use the :ta command without specifying a tag, vi will jump to the last tag that you specified.

If you have set up a tag file for a group of programs, you can start vi and specify that you want to edit the file that holds a particular function. Use the option -t followed by the name of the function, instead of a file name. For example, to edit the file that contains the function sendmsg,

enter:  vi -t sendmsg

vi will use the tag file to determine what file to edit. When vi starts, it will move the cursor to the first line of that function.

To create a tag file, use the ctags program. (ctags stands for “create tags.”) This program is easy to use. It is described in the Xenix Software Command Reference Manual—a part of the software development system. ctags will create a tag file named tags in your working directory.

Here is a summary of the commands in this section. tagname stands for the name of an entry in the tag file.

:ta tagname  jump to function named tagname
:ta          jump to previous tagname

To start vi and edit a particular function that is described in the tag file,

enter:  vi -t tagname
CHAPTER
14
CONTROLLING AND
COMBINING COMMANDS

INTRODUCTION
THE EXECUTION OF PROGRAMS
CONTROLLING PROCESSES
CONTROLLING INPUT AND OUTPUT
COMBINING PROGRAMS
INTRODUCTION

One of the reasons that Xenix is so powerful is that it is flexible. This chapter covers the basic concepts that will allow you to exploit this flexibility. In particular, you will learn how to control the execution of programs and their input and output and how to combine commands to create your own tools.

New Words in This Chapter

PROCESS
the event of running a program

PROCESSID
a unique identification number, assigned to a process by Xenix; pronounced: "process-eye-dee"

FOREGROUND
describes a process in which the program that is running can read and write to the terminal; such processes are said to be running "in the foreground"

BACKGROUND
describes a process in which the program that is running cannot read and write to the terminal; such processes are said to be running "in the background"

DEAMON
a background process started automatically by Xenix to provide a service of general interest

PRIORITY
a number from 0 to 39 assigned to a process that indicates the preference that the process should receive as Xenix allots the resources of the computer; the lower the number, the greater the preference

STANDARD INPUT
the file from which Xenix commands read information

STANDARD OUTPUT
the file to which Xenix commands write information

REDIRECT
to reassign the standard input or standard output to a particular file
SCRIPT
a text file that contains a sequence of commands to be read by a program

FILTER
any program that reads from the standard input and writes to the standard output

PIPELINE
an arrangement in which data is passed through a sequence of programs

PIPE
the mechanism that connects two adjacent programs within a pipeline; when you create a pipeline, you use the | (vertical bar) character to indicate a pipe; PIPE is also used as a verb, meaning to pass data from one program to another

TEE
a filter that reads data from the standard input, makes a copy of the data, and then writes it to the standard output

THE EXECUTION OF PROGRAMS

Processes

A PROCESS is the event of running a program. The shell starts a process for every program that you execute. For example, when you enter the date command, the shell starts the date program as a separate process. When this program is finished—that is, when it has displayed the time and date—it stops executing and the process ends.

Xenix assigns every process a unique number called a PROCESSID, pronounced: “process-eye-dee.” (ld stands for “identification.”) If you need to refer to a particular process, you can do so by its number. For example, you may want to stop process number 453. When a program is finished executing, the processid disappears along with the process.

Thus, we can fashion a picture of the Xenix universe as a strange world inhabited by userids, files, and processes. The userids correspond to the outside users. The files appear and disappear as they are created and removed. The processes appear transiently—like meteors streaking across the sky—every time a program is executed.
There are two ways in which a process can exist. A process in which the program that is running can read and write to the terminal is called a **foreground** process. If the program cannot interact with the terminal, the process is called a **background** process. At any time, you may have only one active foreground process, but many background processes.

For example, say that you use `vi` to edit a file. Since `vi` uses the terminal, you would be using a foreground process. However, at the same time, you might be running a program that takes a long time to perform some complicated calculations. You could run this program as a background process.

When we speak of processes, we refer to them as executing in the foreground or background. Thus, we can imagine that within the Xenix universe, each userid has a foreground and a background area. Each foreground area can hold one process at a time, while each background area can hold more than one process.

Whenever you enter a command, the shell starts a process to carry out that command. Normally, the shell will start a foreground process. However, if you type an ampersand (`&`) at the end of the command, the shell will start a background process.

Here is a sample. In Chapter 16, you will learn how to use the `sort` program to sort all the lines in a file. Say that you wanted to sort a large file named `bigfile`. You would enter:

```
sort bigfile
```

The shell would start a foreground process to carry out the job. However, since the process is running in the foreground, it ties up your terminal and you cannot execute another command until the `sort` program is finished.

Instead, if you enter:

```
sort bigfile &
```

the shell will run `sort` in the background. Thus, you can use your terminal for other work while the `sort` program is running. Of course, you can only do this with programs like `sort` that will run without intervention.

**Deamons**

Xenix is called upon to provide certain services of general interest that require constant or intermittent attention. In order to perform these
services, Xenix starts certain processes called DEAMONS (note the spelling) that run in the background. Some deamons are always active. Others are started only when the need arises. Once they complete their jobs, they vanish.

Since Xenix is designed to run automatically, you never really have to know which deamons are running or what they are doing. However, it is difficult not to be intrigued by the idea of phantom programs flitting in and out of the Xenix universe, responsible to no one userid.

For your interest, whenever I describe a command that depends on a deamon, I will mention the deamon. (The first one is \textit{lpd}, described in Chapter 15.) If you ever want to see which deamons are active, use the \texttt{ps} command with the \texttt{-e} option (described below).

\section*{CONTROLLING PROCESSES}

\subsection*{Keeping Track of Processes—the \texttt{kill} and \texttt{ps} Commands}

Whenever you run a background process, Xenix displays the processid for that process and then starts the process. For example, say that you enter:

\begin{verbatim}
  sort bigfile&
\end{verbatim}

Xenix might display the processid:

\begin{verbatim}
  538
\end{verbatim}

followed by the shell prompt. (The prompt indicates that the shell is now ready to accept another command.)

In Chapter 3, you learned that you press \texttt{<DEL>} to send the \texttt{intr} (interrupt) signal. The purpose of the \texttt{intr} signal is to stop the foreground process. Thus, the \texttt{<DEL>} key has the effect of stopping a command before it is finished, if the command is running in the foreground.

To stop a background process, you use the \texttt{kill} command. The syntax is:

\begin{verbatim}
  kill processid...
\end{verbatim}

where \texttt{processid} identifies the process you want to stop. For example, to stop process number 538,

\begin{verbatim}
  enter: kill 538
\end{verbatim}

Unless you are logged in as the superuser, the \texttt{kill} command will only allow you to stop processes that belong to your own userid.
To display information about all the active processes, use the `ps` command. (`ps` stands for "process status.") This is the command to use when you need to find out the processid for a particular process.

The `ps` command has 12 options, most of which are of interest only to advanced programmers. Here is the syntax for this command, showing the two most important options:

```
ps [-ef]
```

Usually, you will use the `ps` command with no options. In this case, the command will display one line for each process that is running under your userid. Here is an example. You enter:

```
ps
```

and you see:

```
PID  TTY  TIME  COMMAND
929  co  0:04  sh
538  co  0:07  sort
1013 co  0:03  ps
```

The first column shows the processid; the second column shows the name of your terminal (in this case, `co` means "console"); the third column shows the amount of time that the process has been executing; and the fourth column shows the name of the program that is executing.

The time given in the third column is not the elapsed time since the process has existed; rather it is the number of minutes and seconds that the process has been actively executing. In this example, the sort program has used 7 seconds of active processing time.

The first process listed is the `sh` program. This is the Bourne shell. The second process is the `sort` program, which is running in the background. The third process is the `ps` program itself, which is running in the foreground.

When you use the `ps` command, you will always see at least one entry for a shell. The name of the Bourne shell program is `sh`, the name of the C-Shell program is `csh`, and the name of the visual shell program is `vsh`. Sometimes the shell will start a copy of itself to carry out a command. In this case, you will see more than one shell listed.

Usually, the shell is running in the foreground. However, when you enter a foreground command, the shell pauses and lets the program that carries out the command take over the foreground. When the program (in this case `ps`) finishes, it vanishes. The shell takes over the foreground once again and waits for your next command.

When you use the `ps` command with no options, it only lists the
Controlling Processes

processes running under your userid. If you want to see all the processes running under Xenix, including those that Xenix uses for its own purposes, use the -e option. (e stands for "everyone.") If you want to display more information about each process use the -f option. (f stands for "full description.") See the entry for the ps command in the Command Reference Manual for details.

If you want to see which daemons are currently active, use the ps command with the -e option. The daemons are those processes that have a question mark in the TTY column.

Specifying Priorities—the nice Command

Xenix allows many processes to be active at the same time. However, only one process at a time can actually be executing. Xenix automatically schedules all the processes so that they share the computer. As far as you are concerned, your program runs continuously from start to finish. In reality, Xenix stops and starts your program according to a complicated scheduling system.

For example, say that you start a program that requires 15 seconds of computer time. Xenix will allot those 15 seconds in many short fractions of a second. In between, other programs will have a chance to execute. So although your program only requires 15 seconds, it may take 20 seconds for it to receive that much time.

Xenix assigns each process a number from 0 to 39 called a PRIORITY. Processes with lower numbers receive preference when it comes to allotting the resources of the computer. For example, on a busy system, a process with priority 0 will finish much faster than a process with priority 39. (This can be confusing—just remember, the lower the number, the higher the preference.) Xenix constantly examines and changes priorities to meet the needs of the system.

When you enter a command, it is automatically started at a priority of 20. You can enter a command with a different priority by using the nice command. The syntax is:

\[\text{nice [-value] command}\]

where value is a number between 1 and 19, and command is the command that you want to use. The shell will start the command with a priority of 20+value. If you do not specify value, nice will use 10 and start the command with a priority of 30 (20+10).

For example, say that you want to sort a very large file called bigfile. Running the sort program on such a file can slow down the
whole system. To avoid this, use the `nice` command to start `sort` in such a way that it receives low preference as it executes. If you enter:

```
    nice -19 sort bigfile&
```

the shell will start `sort` in the background with a priority of 39 (20+19). This avoids a slowdown because Xenix gives preference to the other processes. Whenever you need to run a time-consuming program, run it in the background with a high priority (low preference) so as not to tie up the system.

The `nice` command usually only allows you to start a program with less than the normal preference. If you are the superuser, `nice` will let you specify a negative number for `value`. This has the effect of starting the command with a lower priority, giving it a higher preference.

For example, say that you want to sort the file `bigfile` in the background with the lower than usual priority of 5. Log in as superuser and enter:

```
    nice --15 sort bigfile
```

In this case, you need two minus signs: one to indicate an option and one to indicate a negative number. The shell will start `sort` with a priority of 5 (20-15). This will cause it to execute faster than if it started at the normal priority of 20.

### Controlling the Execution of Commands

To execute a command only if the previous command was successful, enter the two commands together, separated by `&&`. For example,

```
    sort temp && mv temp master.file
```

In this example, the shell will not execute the `mv` command if the `sort` program encountered some error and could not finish successfully. This prevents the file `master.file` from being replaced by a bad file.

To execute a command only if the previous command failed, enter the two commands together, separated by `||`. For example:

```
    sort temp || echo sort command failed
```

In this example, the `echo` command displays the message `bad sort command` only if the `sort` command failed.

You can use `&&` and `||` more than once on a command line. For example, say that `cmd1`, `cmd2`, `cmd3`, and `cmd4` are four Xenix commands. You want to execute each one in sequence, but only if the
previous commands were successful. Use:

```bash
cmd1 && cmd2 && cmd3 && cmd4
```

In Chapter 20, you will learn how to create files of Xenix commands that you can execute automatically. At that time, you will find `&&` and `||` especially useful.

**CONTROLLING INPUT AND OUTPUT**

**Standard Input and Standard Output**

STANDARD INPUT refers to the file from which Xenix commands read information. STANDARD OUTPUT refers to the file to which Xenix commands write information. Xenix automatically assigns the standard input to the special file that represents the keyboard and the standard output to the special file that represents the display. In other words, Xenix commands read from the keyboard and write to the display.

Xenix allows you to reassign temporarily the standard input and output. For example, the `lc` command will display a list of files on the standard output. Usually, this is the display screen. However, if you want the output to go to a text file instead, you can reassign the standard output to that file for the duration of the command. This is an easy way to create a text file that contains a list of your files.

Similarly, you can reassign the standard input. For example, you can create a text file that contains commands for the `ex` editor. You can then start `ex` but reassign the standard input to that file. Instead of reading commands from the keyboard, `ex` will read them from the file. (This is explained in more detail in the next section.)

When you reassign the standard input or output like this, we say that you REDIRECT it. To redirect the standard output for the duration of a command, type a `>` character followed by a file name. For example:

```bash
lc >file.list
```

The shell will execute the command and redirect the standard output to the file after the `>` for the duration of the command. In this example, the output is directed to the file `file.list`. This command creates a file that contains a list of all the files in the working directory.

To redirect the standard input for the duration of a command, type a `<` character followed by a file name. For example:
ex accounts < ex.script

The shell will execute the command and redirect the standard input to be from the file after the < for the duration of the command. In this case, the shell starts the ex editor and redirects the standard input to be from the file ex.script. Once ex finishes, the standard input returns to the keyboard.

When you redirect the standard output to a text file, the output replaces whatever was already in the file. If you want to add the output onto the end of the file, use two > characters instead of one. For example:

```
ls book > file.list
ls letters >> file.list
ls documents >> file.list
ls programs >> file.list
```

The first command creates a list, in the file file.list, of the names of all the files in the directory book. The second command adds the names of the files in the directory letters to the end of the list in file.list. The next two commands do the same for the names of the files in two other directories.

Using a Script

Any program that reads commands from the standard input can just as easily read from a text file as from the keyboard. A text file that contains a sequence of commands to be read by a program is called a SCRIPT.

In Chapter 13, you learned that you can use the :so command with the vi editor to read an editing script consisting of ex commands. To do this, you have to start vi and then enter the :so command. (Note: Only ex commands, not vi commands, can be read from a script.)

This idea can be expanded. Create an editing script that contains all the ex commands necessary for a complete editing session. You can then enter a command to start the editor, but with the standard input coming from the editing script. This allows you to automate a complicated set of editing commands.

As you learned in Chapter 10, the vi and ex editors are really different faces of the same program. When you start the program by using the vi command, you get a form of the editor that acts as a screen editor.
You can also start the editor by using the `ex` command. The syntax for this command is almost the same as for the `vi` command. When you start the editor with the `ex` command, you get a line editor rather than a screen editor.

If you want to use the editor from the keyboard, you start it with the `vi` command. However, when you want to use an editing script, you must start the editor with the `ex` command. (This is because an editing script can only contain `ex` commands.) For more information, see the entry for the `ex` command in the Command Reference Manual.

Here is an example of a simple editing script. (Note: When you create an editing script, you do not put a colon in front of each command.)

```
%s/Diane/Dianne/g
w
q
```

The first command changes all occurrences of the string `Diane` to the string `Dianne`. The second command saves the editing buffer. The third command stops the editor.

If this script is in the file `ex.script` and you want to use it on the file `test`,

```
enter: ex test < ex.script
```

As I mentioned earlier, you can use scripts with any program that reads commands from the standard input. The `bc` (calculator program) works this way. For example, say that you create a file `calculation` that contains:

```
100-45
```

and you enter the command:

```
bc < calculation
```

`bc` will read the script, perform the addition, and display:

```
55
```

Of course, you can create much more complex and useful scripts for `bc`.

If you want, you can redirect both the standard input and the standard output. For example, if you enter:

```
bc < calculation > sum
```

`bc` will follow the script in the file `calculation` and write the output to the file `sum` (which you can display using the `more` command).
Since the shell itself is a program that reads commands from the standard input, you can create scripts that contain Xenix commands. You will learn how to do this in Chapter 20.

COMBINING PROGRAMS

Filters

Any program that reads from the standard input and writes to the standard output is called a FILTER. Most Xenix commands that use input and generate output can act as filters.

Some programs are always filters. For example, the bc program always reads from the standard input and always writes to the standard output.

Some programs can act as filters in certain circumstances. For example, when you use the more program you can specify the names of one or more files. If you do, more will display the files on the standard output. However, if you do not specify a file name, more will read from the standard input and write to the standard output. In other words, if you do not specify a file name, more will act as a filter.

Some programs are never filters. For example, the lc command can never be a filter because it does not read any input.

Combining Programs by Using Pipes

The importance of filters is that they can be combined. Data can be passed from one filter to the next, while each filter acts as a tool that does a particular job. Such an arrangement is called a PIPELINE (although an assembly line might be a more apt analogy).

Here is an example of how you might use a pipeline: Say that you have a text file in which each line contains a name, address, and phone number. You want to produce a sorted list of all the lines that contain an address in Buffalo. You can connect two filters in a pipeline to process the file. The first filter selects the required lines from the file and the second filter sorts the lines.

Xenix takes care of all the details of passing the data from one program to the next. The mechanism that connects two programs in this manner is called a PIPE. Thus, a pipeline is a sequence of programs.
connected by pipes.

The word "pipe" is sometimes used as a verb to mean passing data from one program to the next. For example, we might say that to display the output of a program, you pipe the output to the more program.

To use a pipeline, enter the names of the programs that you want to connect, separated by a \| (vertical bar) character. Here is an example that involves the grep, sort, and more filters (grep and sort are explained in Chapter 6).

Say that mail.list is a text file in which each line contains a name, address, and phone number. The commands:

```
grep 'Buffalo' mail.list
```

selects all the lines that contain the string Buffalo. The sort program will read lines from the standard input, sort them, and then write the result to the standard output. The more program will read lines from the standard input and display them, one screenful at a time.

To display a sorted list of all the lines in mail.list that contain the string Buffalo,

```
enter: grep 'Buffalo' mail.list \| sort \| more
```

The | character tells the shell to connect the standard output of one program to the standard input of the next (much as an actual pipe connects two parts of a plumbing system). You could simulate this arrangement yourself by using temporary files as follows

```
grep 'Buffalo' mail.list > temp1
sort < temp1 > temp2
more < temp2
rm temp1 temp2
```

The output of grep is redirected to the temporary file temp1, which is used as input for sort. The output of sort is sent to temp2, which is used as the input for more. Once the work is finished, the temporary files are erased. This is similar to what Xenix does for you whenever you use a vertical bar to indicate a pipe.

When you use a pipeline, all but the first and last programs must be filters so that they can read from the standard input and write to the standard output. The only requirement for the first program is that it must write to the standard output. Similarly, the last program only needs to read from the standard input.

Consider this example:

```
ls \| wc -l
```
The first command lists the names of each file in your working directory; each name is on a separate line, and the lines are written to the standard output. Usually, the standard output is the screen, so the list of names is displayed.

In this case, however, the output of `ls` is sent to the `wc` program. The `wc` (with the `-l` option) counts the number of lines and displays that number on the standard output. Thus, the above pipeline has the effect of displaying the number of files in your working directory.

Strictly speaking, `ls` is not a filter because it does not read from the standard input.

Xenix comes with a variety of programs to process data. You are already familiar with some of them, and you will learn about others in Chapter 15. Once you do, you will be able to use pipes, along with redirection of standard input and output, to build yourself all kinds of specialized tools.

### Tees

A TEE is a filter that reads data from the standard input, makes a copy of the data, and then writes it to the standard output. In other words, a tee sits in a pipeline, copies whatever data it receives, and then passes the data along without changing it. A tee allows you to copy the data in a pipeline without disturbing the flow. (In a plumbing system, a pipe that has a similar function might look like the letter `T`; thus the name “tee.”)

To create a tee, use the `tee` command. Here is the syntax (with only the most useful option):

```
    tee [-a] file...
```

`file` is the name of the text file which will hold a copy of the data that passes through the tee. If you specify more than one file name, `tee` will make more than one identical copy. Normally, the output of `tee` will replace whatever was in the file, if the file already exists. The `-a` option specifies that the data is to be added on to the end of the file. (a stands for “append.”)

Here are some examples that use the pipeline from the last section.

```
grep 'Buffalo' mall.llst | sort | more
grep 'Buffalo' mall.llst | tee buf.llst | sort | more
grep 'Buffalo' mall.llst | tee buf.llst usa.list | sort | more
grep 'Buffalo' mall.llst | tee -a master.llst | sort | more
```

The first pipeline displays a sorted list of all the lines in the text file
**Combining Programs**

that contain the string **Buffalo**. The second pipeline is the same, except that it copies the output of **grep** (the unsorted list) to the file **buf.list**. The third pipeline makes two copies of the unsorted list. The fourth pipeline adds the output of **grep** to the end of the file **master.list**.

The Simplest Possible Filter—the **cat** Command

The **cat** command has two forms. One form acts as the simplest possible filter: It passes data from the standard input to the standard output, without modifying the data. The other form of the **cat** command combines files. (This is where the name comes from: **cat** stands for “concatenate.”)

When you use **cat** as a filter, the input and output are specified using redirection. Thus, the syntax is simply:

```
cat
```

Here is an example. The command:

```
cat < old.file > new.file
```

 copies **old.file** to **new.file**, the same as:

```
cp old.file new.file
```

As you remember, when you don’t redirect the standard input, it comes from the keyboard. Thus, the command:

```
cat > stuff
```

reads from the keyboard and writes to the text file **stuff**. This is a quick and easy way to create a file. (Of course, unless the file is short, you are better off using **vi**.)

For example, if you want to create a file named **stuff** that is only a few lines long,

enter:  
```
cat > stuff
```
type the lines, and then press **CTRL-d**. (Remember, **CTRL-d** sends the **eof** character that indicates the end of a file.) If you want to add a few lines to the end of the file **stuff**,

enter:  
```
cat >> stuff
```
type the lines, and then press **CTRL-d**.

You can also redirect the standard output to a special file; that is, you can send the output to a device. In Chapter 15 you will learn how to use **cat** to copy information directly from the keyboard to the printer.
Combining Files—the **`cat`** Command

The second form of the **`cat`** command combines a group of files and writes the result to the standard output.

The syntax is:

```
cat file...
```

where **`file`** is the name of a text file or a special file that can supply input. Here is an example that combines the text files **`sales`, `distributors`,** and **`suppliers`** into one large file named **`accounts`**.

```
cat sales distributors suppliers > accounts
```

Here is an example that counts the number of words in all three files:

```
cat sales distributors suppliers | wc -w
```

**`cat`** processes each file, in the order you specify, by reading the information, line by line, and writing it to the standard output. If you do not redirect or pipe the output, **`cat`** displays it on your terminal. Thus, you can use **`cat`** to display a file. For example,

```
cat sales
```

displays the file **`sales`**. You might ask, Would I ever want to use **`cat`**, instead of **`more`**, to display a file? The answer is no.

When Unix was first developed, **`cat`** was the program that people used to display files. However, **`cat`** displays the lines so fast that you have to keep pressing <CTRL-s> and <CTRL-q> to pause and restart the display. This is a major annoyance. **`more`** was developed at Berkeley, specifically to replace **`cat`** as the program for displaying files.

So, when you want to combine files, use **`cat`**. When you want to display files, use **`more`**. If you ever need to display a group of files as one long file, combine them with **`cat`** and then pipe the result to **`more`**. For example,

```
cat file.1 file.2 file.3 | more
```

combines and displays three files.
CHAPTER 15
READING AND WRITING TO DEVICES

INTRODUCTION
USING SPECIAL FILES
PRINTING
USING DISKETTES
INTRODUCTION

This chapter explains how to read and write to the various devices that are connected to your computer. You will learn how to access the console, other terminals, and the printer. The chapter ends by showing you how to prepare and use diskettes.

New Words in This Chapter

PRIMARY DISPLAY
the display that is used as the output device for the console

PRINT QUEUE
a list of files waiting to be printed

FILE SYSTEM
a group of files organized as a tree

MOUNT
to connect a file system to the main tree

DISMOUNT
to disconnect a file system from the main tree

HIGH-CAPACITY DISK DRIVE
a device that uses diskettes that contain up to 1200 kilobytes of data

DOUBLE-SIDED DISK DRIVE
a device that uses diskettes that contain up to 360 kilobytes of data

FORMAT
to prepare a new diskette

USING SPECIAL FILES

Reading and Writing to Terminals

As you learned in Chapter 7, there are three types of files: ordinary files (which can be text files or executable binary files), directories, and special files. A special file represents a physical device. To read or write to a device, you read or write to its special files.
Xenix has many special files. They are described in Section 2 of the Command Reference Manual. Most of them are useful only to advanced programmers. In the next few sections, I will describe those special files that are of general interest.

The special files that you are likely to use are those that represent the terminal and those that represent the printer.

All special files are in the /dev directory. The file that represents your terminal is /dev/tty. This file can be used for both input and output. When you write to /dev/tty, it refers to the display. When you read from /dev/tty, it refers to the keyboard.

For example, the following command displays the file info on the display screen of your terminal:

```
more info > /dev/tty
```

This has the same effect as:

```
more info
```

This command copies information from the keyboard of your terminal to the file info:

```
cp /dev/tty info
```

This command has the same effect as:

```
cat > info
```

If you want to send information to a particular terminal, you can use /dev/console, /dev/tty00, or /dev/tty01. For example, if you have a short message in the file message, you can display it on tty01 by entering:

```
cp message /dev/tty01
```

Of course, it would probably be less startling to use the write command.

Reading and Writing to the Console

Xenix will allow you to use both a color and a monochrome display with the IBM PC AT. When you set up your computer, you select one of the displays as the PRIMARY DISPLAY, by setting a switch inside the computer and using the setup program that comes on a diskette in the Guide to Operations Manual.

When you are logged in to the console, the primary display is the one that receives your standard output. You can refer to it either as /dev/tty or /dev/console. If you want to send information to the other
display, you can refer to it explicitly as either /dev/color or /dev/monochrome.

For example, say that you are using the console of the IBM PC AT with two displays, and the monochrome display is the primary display. If you want to display the file info on the primary display,

enter: more info

If you want to display the file on the other display,

enter: more info > /dev/color

Here is how this might come in handy. Say that you are writing an essay and that you have some notes in a file called notes. When you use vi to write the essay, vi will use the primary (in this case, monochrome) display. Before you start, you can display the notes on the other display by entering:

more notes > /dev/color

Then, as you are editing with the primary (monochrome) display, you can refer to the information in notes which is still on the other (color) display.

One word of caution: It is a quirk of Xenix that you must refer to the primary display as /dev/tty or /dev/console and not as /dev/color or /dev/monochrome. For example, if the monochrome display is the primary display and you try to refer to it as /dev/monochrome, Xenix will display:

/dev/monochrome: cannot create

It is only when the monochrome display is not the primary display that you refer to it as /dev/monochrome.

PRINTING

Writing to the Printer

Xenix will let you use up to three printers. The special files /dev/lp0 or /dev/lp1 refer to printers that are connected to serial/parallel adapters. The special file /dev/lp2 refers to a printer that is connected to a monochrome and printer adapter. (lp stands for "line printer.") If you
only have one printer, it will most likely be connected to a monochrome and printer adapter, so you will refer to it as /dev/lp2.

One way to print information is to write it to the special file that represents a particular printer. For example, to print the file info on /dev/lp2 you can use:

```
    cp info /dev/lp2
```

As a convenience, the special file /dev/lp serves as a link to whichever of the other printer files represents your main printer. You specify this printer when you install Xenix (see step 6 in the Installation Guide).

For example, if your main printer is /dev/lp2, the installation program establishes the proper link by executing:

```
    ln /dev/lp2 /dev/lp
```

As a general rule, you will always refer to the printer as /dev/lp, unless you do not want your output to go to the main printer. For example, to print the file info you could use:

```
    cp info /dev/lp
```

Another way to print is by redirecting the standard output to the printer. In Chapter 5, you learned how to use the banner program to display words in large letters. You can make a printed sign by redirecting the output of banner to the printer. For example, to make a sign that reads "Wanted Dead or Alive,"

```
    enter: banner Wanted Dead or Alive > /dev/lp
```

If you decide that you want to change the main printer, you do not need to reinstall Xenix. Simply log in as superuser and link the file /dev/lp to the printer file that you want. For example, if you want /dev/lp1 to be the main printer,

```
    enter: ln /dev/lp1 /dev/lp
```

(Many systems have only one printer. In that case, /dev/lp will always be linked to the same printer.)

Here is one last example that you should make sure you understand.

```
    cp /dev/tty /dev/lp
```

This command copies information from the keyboard to the printer, effectively turning your computer system into a very expensive typewriter. (If you try this command for yourself, remember to press <CTRL-d> when you are finished to indicate the end of the file.)
Printing a Text File—the lpr Command

In the last section, you learned that you can print information by writing it to the special file that represents the printer. There is an important problem with this method: You cannot send information to the printer if it is already printing something.

If you have more than one file to print, or if more than one person wants to use the printer, there needs to be some sort of mechanism to schedule the different requests. Xenix uses a daemons named lpd to do just that. (lpd stands for "line printer daemon.")

The lpd daemon maintains a list of files that are waiting to be printed. This list is called the PRINT QUEUE. Files in the print queue are printed on a first come, first served basis.

To add a file to the print queue, use the lpr command. (Note that the name of the command is lpr, for "line printer." lpd is the name of the daemons.) The syntax is:

```
 lpr [ -crm ] [file...] 
```

For example, to print the files report, essay, and list,

```
 enter: lpr report essay list
```

It would be wasteful for the lpr command to make a copy of each file that you want to print. Instead, each entry in the print queue is really a link to the file. This means that if you add a file to the print queue, you should be careful not to change the file before it has been printed.

If this is inconvenient, you can force lpr to make a temporary copy of the file for the print queue. To do this, use the -c option. (c stands for "copy.") lpr will send the copy, rather than the original file, to the print queue. The lpd daemon will erase the copy once it has been printed.

If you use the -r option, the lpd daemon will erase the original file after it has been printed. (r stands for "remove.") This is useful when you want to print a file that you do not need to keep.

The -m option causes the lpd daemon to send you a message via the mail program when the printing is complete. (m stands for "mail"; the mail program is explained in Chapter 8.)

If you do not specify a file name, the lpr command will take its input from that standard input. This means that you can use lpr as a filter. For example, to print a full description of all the files in your working directory,

```
 enter: lc -l | lpr
```

To combine the files old and new and then print them as one large file,
When more than one user sends files to the same printer, it is easy to mix up the output. For this reason, the lpd daemon prints identification pages at the beginning of each file. Each page has the name of the userid that printed the file, followed by the time and date. lpd uses the banner program to print the name of the userid in large letters to make identification easy.

The number of identification pages is controlled by the file /etc/default/lpd. This file contains one line of the form:

```
BANNERS=n
```

where n is the number of identification pages. Normally, n is set to 1 or 2. If you want to change the number of identification pages, log in as the superuser and use vi to edit the file /etc/default/lpd. Change the value of n to whatever you want. If you are the only person who uses the printer, you will probably want to set n to zero to save time and paper.

The lpd daemon prints files in the order that they were added to the print queue. The rule is: first in, first out. You should be aware that there is no easy way to remove a file from the print queue; nor is there an easy way to rearrange the order of the queue. For these reasons, users on small systems sometimes prefer to send files directly to /dev/lp.

Preparing a File to Be Printed—the pr Command

Sometimes you want to print a file, line by line, exactly as it is. More often, it is convenient to prepare the file in some way. Xenix provides the pr program to do this for you.

The pr program breaks a file into pages suitable for printing. (pr stands for “print.”) The pr program has many options, allowing you a great deal of control over the format of the pages. In this section, I will discuss only the more important options.

The syntax for the pr command is:

```
pr [+page] [-dnt] [-h title] [file...]
```

where page is a page number; title is a string of characters; and file is the name of a text file that you want to print. pr will prepare each file that you specify. You have a choice as to what you can do with the output from pr: print it, save it, or display it.

Here are some examples that prepare the file report to be printed: The first two commands print the file right away. The third command
saves the prepared file under the name `report.p`. The fourth command displays the prepared file at the terminal. The last command saves the prepared file and displays it at the terminal.

```bash
pr report | lpr
pr report > /dev/lp
pr report > report.p
pr report | more
pr report | tee report.p | more
```

Left to itself, `pr` will use single spacing and start with the first page. Each page will begin and end with five extra lines. The top five lines will contain the time and date, the name of the file, and the page number; the bottom five lines will be blank. The various options let you change this format.

The `-d` option gives you double spacing. (d stands for "double.") The `-n` option numbers each line of the file. (n stands for "number.") The `-h` option lets you specify a title that is to be used instead of the file name at the top of each page. To use this option, specify the title after the `-h` (h stands for "header.") The `-t` option tells `pr` to omit the extra five lines at the top and bottom. If you want to start at a page other than the first, specify `+` followed by the page number.

Here are some examples, all of which print the file `report` by piping the output of `pr` to `lpr`. The first example uses double spacing. The second example numbers each line of the file. The third example specifies a string that should be used at the top of each page, instead of the file name. The fourth example does away with the extra five lines at the top and bottom. The last example omits the first four pages of the file by starting at page 5.

```bash
pr -d report | lpr
pr -n report | lpr
pr -h ' Fiscal Year-End Report ' report | lpr
pr -t report | lpr
pr +5 report | lpr
```

Notice, in the second example, that the string is enclosed by single quotes. This is necessary because the string contains space characters and a special character.

If you do not specify a file name, `pr` will read from the standard input. Thus, you can use `pr` as a filter. Here are two such examples: The first combines three files and then prepares and prints them. The second produces a printed list of all the files in the entire Xenix system.

```bash
cat file1 file2 file3 | pr | lpr
lc -R / | pr | lpr
```
(The `lc` command starts from the root (/) directory and processes every sub-directory.)

USING DISKETTES

File Systems

A FILE SYSTEM is a group of files organized as a tree. You must create a file system on each diskette that you want to use.

As you learned in Chapter 7, Xenix organizes files into one main tree structure. To use a diskette, you must connect its file system with the main tree. This is like grafting a small tree onto one of the branches of a large tree. When you are finished using the diskette, you must disconnect its file system from the main tree.

When you connect a file system to the main tree, we say that you MOUNT the file system. When you disconnect a file system, we say that you DISMOUNT it. The names come from the days when auxiliary file systems were stored on reels of magnetic tape. The reels had to be physically mounted on the tape drives.

Xenix allows you to work with the main tree as if it were one complete structure. However, it is really made up of two separate file systems (named root and usr) that are stored on the fixed disk. Every time you start Xenix, it builds the tree by mounting the two file systems. As well, every time you use a diskette you mount a file system.

Xenix handles all the details automatically. However, there is one important concept that you should understand. When you write information to a file, Xenix must actually transfer the information to a file system residing on a physical device. Xenix does not necessarily do this right away; sometimes Xenix will wait for a more convenient time.

To understand this, think of what happens when you return books to a library. It would be a waste of time for the librarian to put each book away as it came in. Instead, the librarian may let the books collect and shelve them every hour. This practice has two advantages: First, the librarian can get on with other work; and second, if someone needs a book that is waiting to be shelved, it can be found immediately. In the Xenix library, it often happens that the librarian needs the most recently returned books, so this system works well.

The fact that Xenix does not necessarily perform all its I/O (input/output) immediately will not affect your work. The librarian will
make sure that the right book is in your hand at the right time. However, the issue does come up in a few ways that you can notice.

Remember, Xenix must maintain two file systems on the fixed disk. This means that every now and then Xenix must check the status of these systems and bring them up to date.

Each disk drive has a light that goes on whenever the drive is in use. If you are using the console of an IBM PC AT, you will notice that the light for the fixed disk goes on at odd times, even when you are not doing anything. All it means is that Xenix is accessing the disk in order to maintain the file systems properly. When you use a diskette, you will see that the light on the diskette drive goes on at unexpected times. Again, it is just Xenix doing I/O in its own way.

This is why you can’t just turn off the computer when you want to stop Xenix. There may be information that Xenix has not yet written to the disk. When you use the haltsys commands, Xenix has a chance to clear up the pending I/O and shut down the system gracefully. If you do not use this command, you run the risk of losing data. This would be like clearing out the librarian’s desk before he or she had a chance to put all the books away.

When you use a diskette, it is important to realize that you cannot just take it out of the drive when you are finished. You must dismount it first. This gives Xenix a chance to bring the diskette’s file system up to date.

Special Files for Diskettes

When you run Xenix on the IBM PC AT, the computer may have either one or two diskette drives. There are two types: A HIGH-CAPACITY DISK DRIVE uses diskettes that hold 1200 kilobytes of data; a DOUBLED-SIDED DISK DRIVE uses diskettes that hold 360 kilobytes of data.

The top drive on the PC AT is always a high-capacity drive; it is referred to as drive 0 (zero). The optional drive can be either type; it is referred to as drive 1 (one). If you have two drives of different types, remember to use the correct diskettes with each one.

Xenix has a number of special files that you use to access diskettes. Most of the time, you will refer to the diskettes in drives 0 and 1 as /dev/fd0 and /dev/fd1, respectively. (fd stands for “floppy disk,” another name for diskette.)

When you prepare a diskette by using the format program (see next
To format a diskette, use the **format** command. The syntax is:

```
format /dev/rfd0  /dev/rfd1
```

Specify the special file that refers to the drive that is holding your diskette. For example, to format a high-capacity diskette in drive 0, insert the diskette (remember to turn down the lever on the drive), and enter:

```
format /dev/rfd0
```

Normally, you only need to format a diskette once. However, there are two cases in which you would format a diskette again: first, if you want to use the diskette with another operating system (each operating system uses its own particular format); second, if a diskette does not work properly, you can try to format it again before you throw it away. However, this is a measure of last resort because it will erase all the data on the diskette.

**Making a File System—the mkfs Command**

Before you can use a formatted diskette, you must make a file system on it. To do this, use the **mkfs** command. (**mkfs** stands for “make a file system”; this command is easy to misspell so specify carefully.) This command is a complex one with various options. However, the syntax
you will normally use is:

/etc/mkfs device size

where device is the special file that refers to the drive that holds the diskette, and size is the storage capacity of the diskette in kilobytes. For high-capacity diskettes, size is 1200; for double-sided diskettes, size is 360.

(Notice that you must refer to the command as /etc/mkfs, not as mkfs. See the section on setting the search path in Chapter 21.)

To make a file system on a high-capacity diskette in drive 0, use:

/etc/mkfs /dev/fd0 1200

To make a file system on a high-capacity diskette in drive 1, use:

/etc/mkfs /dev/fd1 1200

To make a file system on a double-sided diskette in drive 1, use:

/etc/mkfs /dev/fd1 360

When you make a file system, it erases any information that was previously on the diskette. If there is already a file system on the diskette, mkfs will display a message such as:

mkfs: /dev/fd0 contains data. Overwrite? (y/n):

If you want to replace the file system with a new one, enter y (for “yes.”) If not, enter n (for “no”) and the diskette will not be changed.

Checking a File System—the fsck Command

If something goes wrong with your diskette, you can check the file system by using the fsck command. (fsck stands for “file system check”; the name is easy to misspell so type carefully.) This command is a complex one with various options. However, the syntax you will normally use is:

fsck device

where device is the special file that refers to the drive that holds the diskette. For example, to check the file system on the diskette in drive 0, use:

fsck /dev/fd0

fsck will go through a five-step process that examines the various parts of the file system. If all goes well, you will see a message such as:
**Phase 1 - Check Blocks and Sizes**
**Phase 2 - Check Pathnames**
**Phase 3 - Check Connectivity**
**Phase 4 - Check Reference Counts**
**Phase 5 - Check Free List**
7 files 177 blocks 10168 free

If `fsck` finds something wrong, it will ask you for permission to make repairs. Enter `y` for “yes” or `n` for “no.” If the file system is damaged, you may lose data. However, `fsck` will always recover as much data as possible, so there is rarely a reason to answer “no.”

### Mounting a File System—the `mount` Command

Before you can use a file system on a diskette, you must use the `mount` command to mount the file system. This connects it with the main Xenix file structure. The syntax is:

```
/etc/mount [[-r] device directory]
```

where `device` is the name of the special file that refers to the drive that holds the diskette, and `directory` is the name of the directory in the main tree to which you want to attach the file system.

(Notice that you must refer to the command as `/etc/mount`, not as `mount`. This is explained in the section on setting the search path in Chapter 21.)

Before you mount a file system, you must create a directory to which the file system will be attached. At the time you use the `mount` command, the directory must meet the following criteria:

- it must be empty
- it must not be the working directory for any userid

If someone is using the directory as a working directory, `mount` will tell you that the directory is busy. Usually this is because you have inadvertently made it your own working directory.

Here is an example of how to use the `mount` command. You want to mount the file system on a diskette in drive 0 and connect it to the directory `/usr/acme/accounts`. First, make the directory (you only have to do this once):

```
mkdir /usr/acme/accounts
```

Next, mount the file system:
/etc/mount /dev/fd0 /usr/acme/accounts

Note: When you use mount, you must always refer to a directory by its full pathname. For example, even if your working directory is /usr/acme, you must still refer to the mounting directory as /usr/acme/accounts; not as accounts.

Xenix connects the file system so that its root is in this directory. For example, if the file system contains the files sales, repairs, and investments, you can now refer to them as:

/usr/acme/accounts/sales
/usr/acme/accounts/repairs
/usr/acme/accounts/investments

When you do, Xenix recognizes the directory accounts as being on a diskette. Xenix will automatically access the diskette to read the files.

If you write to a file under this directory, Xenix will write to the diskette. For example, to copy the file report from your working directory to the diskette, use:

cp report /usr/acme/accounts

This is one way to keep extra copies of important files on a diskette. (See Chapter 19 for a more complete discussion.)

Once a file system is mounted, you can use it with any of the file commands. For example, to change your working directory to the root of the file system in the last example, use:

cd /usr/acme/accounts

If you use the -r option, mount will connect the file system in such a way that you can read it but not write to it. (r stands for "read-only.") If you are using a diskette that is write-protected, you must use this option or there will be errors when Xenix tries to update the file system. (A write-protected diskette does not have an uncovered notch in its front, upper-right corner.)

If you use the mount command with no arguments, it will display a list of all the currently mounted file systems. You will see the names of the two file systems that make up the main part of the tree (root and usr), along with any file systems that you have mounted.

Dismounting a File System—the umount Command

When you are finished using a file system, use the umount command to dismount it. (Note the spelling of this command.) If you are using a diskette, you must dismount the file system before you remove the
diskette from the drive. The syntax is:

/etc/umount device

where device is the name of a special file that refers to the drive that holds the diskette.

(Notice that you must refer to the command as /etc/umount, not as umount. See the section on setting the search path in Chapter 21.)

For example, to dismount the file system on the diskette in drive 0, use:

/etc/umount /dev/fd0

The only rule is that no userid can have a working directory that is on the diskette. For example, say that you mount a file system by using:

/etc/mount /dev/fd0 /usr/acme/accounts

and that you make /usr/acme/accounts your working directory by using:

  cd /usr/acme/accounts

If you try to dismount /dev/fd0, umount will tell you that the device is busy. Change your working directory and try again.

Using a Diskette—Summary

Here is a summary of the procedure for using a diskette. The examples assume that you want to use a high-capacity diskette in drive 0 and that you want to attach the file system to the directory /usr/acme/accounts.

Start by inserting the diskette in the drive. Be sure to turn down the lever.

1. Format the diskette:

   format /dev/rfd0

   (You only need to do this once for each diskette.)

2. Make a file system on the diskette:

   /etc/mkfs /dev/fd0 1200

3. Create a directory in which to mount the file system:

   mkdir /usr/acme/accounts

   (You only need to do this once. You can use the same directory over and over, as long as it is empty.)
4. Mount the file system:

   `/etc/mount /usr/acme/accounts`

   You can now refer to the files on the diskette as if they were in the specified directory.

5. When you are finished with the diskette, dismount the file system:

   `/etc/umount /dev/fd0`

   You can now remove the diskette from the drive.
CHAPTER 16 SEARCHING, SORTING, AND COMPARING

INTRODUCTION
SEARCHING TEXT FILES
SORTING TEXT FILES
COMPARING TEXT FILES
INTRODUCTION

Xenix offers a variety of commands that allow you to sort, search, and compare text files. In this chapter, you will learn the most useful of these commands. You will find them to be important additions to your Xenix toolbox. As you read the examples, notice how the basic operations can be combined to perform many different types of tasks.

New Words in This Chapter

**RECORD**
within a text file, the largest unit of information holding one or more related items of data

**FIELD**
within a record, one item of information

**SORT**
to rearrange the records in a text file according to a specified ordering based on the values of one or more fields in each record

**KEY**
the part of a record that is compared during a sorting operation

**STABLE**
describes a sorting program that never changes the order of records that have equal key values

**UNSTABLE**
describes a sorting program that sometimes changes the order of records that have equal key values

SEARCHING TEXT FILES

Searching for a Pattern—the grep Program

Xenix provides a family of programs to search one or more text files for a specified pattern. These programs are `grep`, `egrep`, and `fgrep`.

When you use these programs, you specify a pattern by using a regular expression. The name `grep` stands for “global, regular expres-
"Searching Text Files" 233

"global" refers to the fact that grep searches the entire file; "print" is used in the Unix tradition of meaning "display." grep searches an entire file and displays all the lines that contain a pattern that matches the regular expression.

The other two programs are variations of grep. egrep stands for "extended grep"; fgrep stands for "fast grep."

There are two ways to use grep. The syntax is:

```
grep [-chlnvy] pattern [file...]  
grep [-chlnvy] -f patternfile [file...]
```

where pattern is a regular expression; patternfile is the name of a text file that contains a list of regular expressions; and file is the name of a text file you want to search.

Usually, you will use the first form of grep without any options. For example, the following command searches the text file addresses and displays all the lines that contain the string Buffalo:

```
grep Buffalo addresses
```

With the first form of the grep command, you specify a regular expression as part of the command. With the second form, you specify -f, followed by the name of a text file that contains one or more regular expressions, each on a separate line. grep will search for lines that contain any of the patterns. For example, say that the file cities contains the lines:

```
Buffalo
Miami
```

The following command searches the text file addresses and displays all the lines that contain either of the strings Buffalo or Miami:

```
grep -f cities addresses
```

The various options let you control the output of grep. The -c option displays the number of lines that match the pattern, rather than the lines themselves.

For example, say that addresses is a text file that contains a list of all your friends' addresses. You want to know how many of your friends live in Buffalo. One way to find out is to search for all the lines that contain the string Buffalo and then use wc to count the number of lines.

```
grep Buffalo addresses | wc -l
```

A simpler way is to use the -c option:

```
grep -c Buffalo addresses
```
When you use **grep** you can specify more than one file to search. For example, say that you have written a book and the chapters are stored in the files `chapter1`, `chapter2`, `chapter3`, and `chapter4`. You forget which chapter explains about anacondas. An easy way to find out is to search the four files for the string **anaconda**.

```
grep anaconda chapter1 chapter2 chapter3 chapter4
```

As with most Xenix commands that use files, you can specify a pattern of file names. The following command is equivalent to the previous one:

```
grep anaconda chapter[1-4]
```

When you search more than one file, **grep** displays the name of the file at the beginning of each line that is selected. Sometimes, all you want to know is which files contain the pattern; you don’t really care to see the actual lines. In this case, use the `-l` option. **grep** will display only the name of each file that contains the pattern.

For example, consider the four files described in the last example. Say that only `chapter1` and `chapter3` contain the string **anaconda**. The command:

```
grep -l anaconda chapter[1-4]
```

will display

```
chapter1
chapter4
```

The `-n` option displays all the selected lines, preceded by a line number. (`n` stands for "number.") The file itself does not need to contain line numbers; **grep** counts for itself as it searches.

The `-v` option displays all the lines in the file that do not contain the pattern. Here is an example: Say that the file `customers` contains one line for each of a company's customers. Somewhere on the line is the string **paid**, if the customer has paid his or her bill. The following command displays a list of all the customers who have paid their bills:

```
grep paid customers
```

The next command displays a list of the customers who have not paid their bills (by searching for the lines that do not contain the string **paid**).

```
grep -v paid customers
```

Normally, **grep** distinguishes between upper and lower case as it compares letters. The `-y` option tells **grep** not to make this distinction. For example, the command:
grep -y the Info

searches the file info. During the search, all the following strings will match the pattern the:

the The tHe thE tHE ThE THE THE

If you use grep, egrep, or fgrep without specifying a file to be searched, they will read from the standard input. Thus, you can use these programs as filters. For example, to display the names of all the files in your working directory that contain the string report, use:

ls | grep report

Here is one more example: When you use the -R option with the lc command (to process all sub-directories), the only lines of output that contain a / character are those that contain the name of a directory. The following command searches for such lines to print a list of all the directories in the entire Xenix system:

lc -R / | grep '/' | pr -h 'Xenix directory list' | lpr

Using Regular Expressions with grep

When you specify a regular expression for grep, you can use almost all of the special symbols that you use with vi and ex. Here is a list of the symbols grep recognizes:

.       matches any one character except a newline
*       matches zero or more of the preceding characters
^       matches the beginning of a line
$       matches the end of a line
[ ]     matches one of the enclosed characters
[^ ]    matches any character that is not enclosed

Inside [ and ], you can use - to indicate a range of single letters or digits. (For example, [A-Z].) \ indicates that the following character is to be taken literally. Unlike with vi and ex, there is no way to match the beginning or end of a word.

The rules for using these symbols are explained in the section on regular expressions in Chapter 9. There will be some examples in this section, but you may want to take a moment to review the section in Chapter 9 before you continue. For simplicity, all the examples in this section will search for one pattern in a text file named info.
When you use special characters to specify a pattern, you should put the entire pattern within single-quotes. For example, to search for the lines that contain `ab`, use:

```
grep ab info
```

To search for the lines that start with `ab`, use:

```
grep 'ab' info
```

You might wonder, Are the quotes really necessary? Not only that, but how is it that the `^` character retains its special meaning inside quotes? Shouldn't the `^` be taken literally? The answer involves a subtle but important point.

Before a command can look at its arguments, those arguments must be processed by the shell. As you know, the shell recognizes certain characters as having special meanings. For example, the `*` character is used to match the names of files. (See Chapter 9.)

The shell looks for and interprets these characters whenever it processes an argument. When you use special characters, you run the risk of having them interpreted by the shell before they even get to `grep`. The single-quotes tell the shell to pass the argument to `grep` literally. Another way of saying this is that you use single-quotes to hide special characters from the shell. (This is true in general, not only with `grep`.)

The reason that `grep` does not treat the special characters literally is that the shell removes the quotes before passing the argument. Thus, in the last example, `grep` sees `^ab` rather than `'^ab'`.

The following command searches for lines that contain any string that starts with an `a` and ends with a `b`:

```
grep 'a.*b' info
```

If you want `grep` to treat a special character literally, simply precede the character with a `\` (backslash). Thus, to search for lines that contain the string `a.*b`, use:

```
grep 'a\.*b' info
```

To search for the string `1\2`, use:

```
grep '1\2' info
```

The egrep Program

The `egrep` program was developed to use in cases where `grep` might be too slow. (`egrep` stands for "extended grep.") The trade-off is that
egrep may require more memory than grep. For most everyday tasks, grep will work fine.

The syntax for the egrep command is:

```
egrep [-chlnv] pattern [file...]
egrep [-chlnv] -f patternfile [file...]
```

where pattern is a regular expression; patternfile is the name of a text file that contains a list of regular expressions; and file is the name of a text file you want to search.

The syntax and options for egrep are the same as for grep, except that egrep does not have the -y option (to ignore the distinction between upper and lower case).

However, aside from speed, egrep has an important advantage over grep in that you can use more complex regular expressions. For this reason, you may occasionally want to use egrep, even if grep is fast enough.

Here is a list of symbols that egrep recognizes within regular expressions. The extra characters are the * and ?.

- . matches any one character except a newline
- * matches zero or more of the preceding characters
- + matches one or more of the preceding characters
- ? matches zero or one of the preceding characters
- ^ matches the beginning of a line
- $ matches the end of a line
- [ ] matches one of the enclosed characters
- [^ ] matches any character that is not enclosed

Inside [ and ], you can use - to indicate a range of single letters or digits. (For example, [A-Z].) \ indicates that the following character is to be taken literally.

Here are some examples of regular expressions using the * and ? characters. The regular expression:

```
[A-Z]x
```

matches an upper-case letter, followed by the character x. For example:

```
Ax  Bx  Cx  Zx
```

The regular expression:

```
[A-Z]*x
```
matches zero or more upper-case letters, followed by the character \texttt{x}. For example:
\begin{verbatim}
x  Ax  Bx  Cx  ABCx
\end{verbatim}

The regular expression:
\begin{verbatim}
[A-Z]+x
\end{verbatim}

matches one or more upper-case letters, followed by the character \texttt{x}. For example:
\begin{verbatim}
Ax  ABx  ABCx  MXYZPTLKx
\end{verbatim}

This last regular expression is equivalent to:
\begin{verbatim}
[A-Z][A-Z]*x
\end{verbatim}

The regular expression:
\begin{verbatim}
[A-Z]?x
\end{verbatim}

matches zero or one upper-case letter, followed by the character \texttt{x}. For example:
\begin{verbatim}
x  Ax  Bx  Cx
\end{verbatim}

\texttt{egrep} allows you to enclose part of the regular expression in parentheses and have it count as one unit. You can use this before the \texttt{*}, \texttt{+}, or \texttt{?} characters. For instance, the regular expression:
\begin{verbatim}
x(123)*y
\end{verbatim}

matches the character \texttt{x}, followed by zero or more occurrences of the characters \texttt{123}, followed by the character \texttt{y}. For example:
\begin{verbatim}
xy  x123y  x123123y  x123123123y
\end{verbatim}

The regular expression:
\begin{verbatim}
x(123)+y
\end{verbatim}

matches the character \texttt{x}, followed by one or more occurrences of the characters \texttt{123}, followed by the character \texttt{y}. For example:
\begin{verbatim}
x123y  x123123y  x123123123y
\end{verbatim}

The last advantage that \texttt{egrep} has over \texttt{grep} is that you can specify more than one regular expression without having to put them in a separate file. To do this, separate the regular expressions either with a \texttt{|} (vertical bar) or a \texttt{\newline} character.

For example, say that you want to search the file \texttt{addresses} for lines that contain either of the strings \texttt{Buffalo} or \texttt{Miami}. One way is to create a
file (call it cities) with the lines:

    Buffalo
    Miami

and use the command:

    egrep -f cities addresses

Using the | or newline characters, we can use either of the commands:

    grep 'Buffalo|Miami' addresses
    grep 'Buffalo<ENTER>Miami' addresses

This feature comes in handy when you need to search for a number of simple strings.

The fgrep Program

The fgrep program is faster than grep and does not need as much memory as egrep. However, fgrep is limited in that it can only search for patterns that match a string exactly; fgrep cannot use regular expressions. For most everyday tasks, grep will work fine.

The syntax for the fgrep command is:

    fgrep [-chnvx] string [file...]
    fgrep [-chnvx] -f stringfile [file...]

where string is a string of characters; stringfile is the name of a text file that contains a list of strings; and file is the name of a text file you want to search.

The syntax and options for fgrep are the same as for grep, except that fgrep has one extra option. The -x option will match only those lines that are exactly the same as the string.

For example, say that the file info contains the lines:

    name
    the name is Henry
    this is the name

The command:

    fgrep name info

will match all three lines. The command:

    fgrep -x name info

will match only the first line.
SORTING TEXT FILES

Introduction to Sorting

Xenix comes with a program that you can use to sort text files. However, before you learn to use this program, there are a number of concepts and terms with which you must become familiar.

Sorting and searching involves processing files of information. Within a file, the information is organized into collections called RECORDS. Each record holds one or more related items of information.

For example, consider a file that holds information about all the employees of a company. The file might be organized so that each record holds the information about one employee. If there are 100 employees, there will be 100 records.

Within each record, the various items of information are called FIELDS. For example, in a file of employee information, each record might contain a field for the social security number, a field for the first name, a field for last name, a field for the home address, and so on.

SORTING a file means rearranging the records based on the values in one or more fields. For example, you might sort a file of employee information based on the last name of each employee. A more technical way of saying this is that you would sort the records according to the value of the “last name” field.

When you sort a file, you often ignore some of the fields. In the above example, you would ignore all the fields except the one that contains the last name. The parts of the record that are actually compared are called KEYS. In this case, the only key is the “last name” field.

When you use a sorting program, you tell it which parts of the record are keys. The program will rearrange the records of the file, based on the key values. You can choose the keys as you want. For example, you might sort the employee file by last name, or you might sort by social security number.

Generally speaking, there are two ways to sort: in alphabetical order and in reverse alphabetical order. But what “alphabet” should you use? It should contain not only the letters but also the numbers and special characters. As well, you will want to distinguish between upper-case and lower-case letters.

In Chapter 7, I explained that Xenix uses a standardized representation of characters called the ASCII code. Within this code, the characters have a particular order. The “alphabet” that Xenix uses for
sorting is the ASCII code, and the order of the alphabet is the order that the characters appear in the code.

I have included a copy of the ASCII code as an appendix to this book. As well, a copy exists within Xenix itself, in the file /usr/pub/ascll (pub stands for "public.") You can display this file at your terminal any time you need to check the position of a character. Use:

`more /usr/pub/ascll`

The characters will be displayed across the screen. Each character will be preceded by a number.

This file really holds two copies of the ASCII code. In the first copy, the numbers represent the ranking of the characters in base 8; in the second copy, the numbers are the ranking in base 16. (If this doesn't mean anything to you, ignore the numbers.)

As a general rule, just remember that the numbers and letters are in order (0-9 and a-z), and that the numbers come before the upper-case letters, which come before the lower-case letters. Thus, 5 comes before R, which comes before r.

The sort Program

The sort program is a powerful tool that can perform several different tasks. First, sort can sort the lines in a file, based on the value of all or part of each line. Second, sort can check a file and tell you if it is in sorted order. Third, sort can take a number of sorted files and merge them into one large sorted file.

Here is the syntax for the sort command. Only the most important options are described.

```
sort [-bcdflmnru] [-tchar] [+position [-position]] [file...]```

where file is the name of a text file that you want to sort. The values of char and position are explained below.

The sort program reads from the file you specify, sorts it, and writes the results to the standard output. For example, the following command sorts the file info and displays the results at the terminal:

```
sort info
```

If the file is large, you can send the output to more.

```
sort info | more
```

If you want to save the result, redirect the output to a text file. The
following command writes a sorted copy of the file info to the file newline.

        sort info > newfile

sort does not change the original file. Unfortunately, if you want to replace the original file with the sorted file, you cannot redirect the output back to the original file. This is because Xenix performs the redirection in such a way that the contents of the file will be lost. For example, if you enter:

        sort info > info

you will end up with an empty file (a possibly upsetting occurrence). Instead, use the -o option, followed by the name of the file to which you want to send the output. This can be any text file, including the original file. (o stands for “output.”)

Here are two examples. The first one replaces info by a sorted copy of itself. The second one is equivalent to redirecting the output to the file newfile.

        sort -o info info
        sort -o newfile info

If you do not specify a file name, sort will read from the standard input. Thus, you can use sort as a filter. For example, to display a sorted list of the userids that are currently logged in,

        enter: who | sort

If you specify more than one file, sort will combine them before it starts to sort. In other words,

        sort file1 file2 file3 file4

is equivalent to:

        cat file1 file2 file3 file4 | sort

GENERAL OPTIONS

sort has a number of options that allow you to control its operation. The -c option checks if the file is in sorted order. Thus, no sorting is actually performed. (c stands for “check.”) If the file is in order, sort will not display any message. If the file is not in order, sort will display the name of the file, followed by disorder, followed by the first line that was out of order.
The `-d` option tells `sort` to consider only letters, numbers, and `space` and `tab` characters during the sorting process. This option ignores all the special characters (including punctuation). (`d` stands for "dictionary," because dictionaries follow similar rules.) For the purposes of sorting, `tab` comes before `space`, and they both come before the numbers and letters.

The `-f` option tells `sort` to consider all upper-case letters as if they were lower case. For example, `Zoo` would normally come before `aardvark` (because `Z` comes before `a` in the ASCII code). When you use the `-f` option, the `Z` is counted as lower case and `aardvark` comes before `Zoo`. (This does not change the original file.)

The `-l` option tells `sort` to ignore certain characters from the ASCII code. (`l` stands for "ignore.") The characters that `sort` ignores are the ones that represent special signals. Normally, these characters will not find their way into a text file so you can safely ignore this option. The strict definition of this option is that it uses only those characters that have octal values from 040 to 176 inclusive. If you don’t understand this, ignore it.

The `-m` option merges several sorted files into one large sorted file. (`m` stands for "merge.") For example, say that the files `january`, `february`, and `march` contain sorted lines. To merge these files into one large file called `first.quarter`,

```
    enter: sort -m january february march > first.quarter
```

The `-n` option sorts numbers by their numeric value, rather than character by character. For example, say that `numbers` is a file in which each line contains a number. Three of the lines are:

```
    99999.99
    1.98
   -1.98
```

The second and third numbers have `space` characters in front of them in order to line up the decimal point. If you enter:

```
    sort numbers
```

`sort` will compare character by character. The result is:

```
    1.98
   -1.98
    99999.99
```

which is inappropriate because a negative number should precede a positive number. However, if you use:

```
    sort -n numbers
```

```
    1.98
   -1.98
    99999.99
```
sort -n numbers

sort puts the characters together to create a numeric value, and the result is:

-1.98
1.98
99999.99

With the -n option, a number can be preceded by any number of space or tab characters and a minus sign. As well, the number may contain a decimal point.

The -r option sorts in reverse order. (r stands for "reverse.")

The -u option eliminates duplicate lines (u stands for "unique.") For example, say that the text file words contains a large number of words, each on a separate line. Many of the words occur more than once. You want to sort and display the words, but you only want to display each word once. Use:

sort -u words

SORTING WITH FIELDS

As you learned earlier, a file that is to be sorted can be considered as a collection of records, each record containing a number of fields. During a sorting operation, the part of the record that is being compared is called a key.

sort considers each line to be a complete record. Another way to put this is that sort processes a file line by line. Unless you specify otherwise, sort considers each line to be one large key. However, sort is capable of recognizing fields within lines and distinguishing between keys and other fields. For example, you can sort a file based on the value of the third and fourth field of each record.

In this section, you will learn how to specify which parts of the lines are to be keys. All the examples will refer to a text file called employees.

Unless you specify otherwise (see below), sort assumes that the fields in a record are separated by either space or tab characters. For example, the following line has three fields:

1234-56-7890  Robert  Prior

The first field has the value 1234-56-7890; the second field has the value Robert; and the third field has the value Prior.

When you sort using fields, you specify the beginning and end of each key in terms of field numbers. One tricky thing is that sort counts
fields starting at zero. Thus, the first field is field 0, the next is field 1, and so on. If you want to refer to the eighteenth field, you must call it field 17.

To define a key, use a + character, followed by the number of the field that begins the key; then, use a - character, followed by the number of the field following the key.

For example, to sort the file `employees` based on the value of the third field, use:

```
sort +2 -3 employees
```

In other words, the key starts at field 2 (the third field) and extends up to, but does not include, field 3 (the fourth field).

The next example uses two keys. One key consists of the fourth, fifth, and sixth fields (fields 3-5); the other key consists of the tenth field (field 9).

```
sort +3 -6 +9 -10 employees
```

If you use a + specification without a - specification, `sort` assumes that the key extends to the end of the line. For example, to define a key that starts at the eighth field (field 7) and extends to the end of the line, use:

```
sort +7 employees
```

When necessary, `sort` allows you to be even more precise. You can start the key at a particular character within a field. At the end of the field number, add a period followed by the position within the field at which the key starts.

For example, say that the first field (field 0) of the `employees` file consists of social security numbers of the form:

```
1234-56-7890
```

To sort according to the value of this field, use:

```
sort +0 -1 employees
```

However, to sort according to the value of the last four characters of the field use:

```
sort +0.9 -1 employees
```

This means that the key starts at nine positions past the beginning of field 0 and continues up to, but does not include, the beginning of field 1. You can use the same type of specification to define the end of a field. For example, the command:

```
sort +0.9 -2.4
```

means that the key starts at nine positions past the beginning of field 0
and extends up to, but does not include, four positions past the beginning of field 2. In other words, the key contains the first three characters of field 2.

When you use more than one key, sort looks at them in the order you specify. Later keys are examined only if earlier keys have different values.

For example, say that the first three fields of the employees file contain a social security number (field 0), a first name (field 1), and a last name (field 2). You want to sort the file by last name. If two last names are the same, you want to compare first names. If both names are the same, you want to compare social security numbers. The following command defines the three keys in order of priority:

```
sort +2 -3 +1 -2 +0 -1 employees
```

The question arises: What does sort do with lines that have equal values on all the keys you specify? The answer is that sort starts from the beginning of the line and looks at other fields. For example, say that the file employees has the following two lines in this order:

```
1234-56-7890 Robert Prior
0000-00-0000 Robert Redford
```

You enter a sort command to use the second field (field 1) as a key:

```
sort+1 -2 employees
```

What happens? Since these two lines have equal values in the second field, sort compares the rest of the fields starting from the beginning of the line. The result is:

```
0000-00-0000 Robert Redford
1234-56-7890 Robert Prior
```

If a sorting program never changes the order of records that have equal key values, we say that the program is STABLE. If a program sometimes changes the order of records that have equal key values, we say that the program is UNSTABLE. The sort program is unstable.

**OPTIONS TO USE WITH FIELDS**

There are two options that are used only when you sort with fields. As well, there are several options that you can use to describe a particular field.

Usually, sort considers fields to be separated by one or more space
or **tab** characters. For example, the line

```
1234-56-7890    Robert    Prior
```

has three fields. However, you may find yourself working with a file in which another character separates the fields. If this is the case, use the `-t` option followed by that character. (*t* stands for “tab.”) Do not put a **space** between the *t* and the character.

For example, say that you have a file called **workers** in which the fields are separated by `$` characters. For instance:

```
1234-56-7890$Robert$Prior
```

To sort the file, using the second field (field 1) as a key, use:

```
sort -t$ +1 -2
```

The `-b` option ignores **space** and **tab** characters at the beginning of each key. (*b* stands for “blank.”) Warning: Owing to a flaw in the program, this option does not always work.

When you define keys, you can specify that one or more options should apply to a particular field. For example, you might want `sort` to ignore **space** and **tab** characters at the beginning of one particular key, instead of every key.

The options that you can apply to individual fields are:

- `b` Ignore beginning **space** and **tab** characters
- `d` Compare only letters, numbers, **space**, and **tab**
- `f` Change upper-case letters to lower case
- `l` Ignore special signal characters
- `n` Use a numerical value
- `r` Sort in reverse

To apply one or more of these options to a particular field, put the letter for those options directly after the specification for the field. You can do this for both `+` and `-` specifications.

For example, the following command sorts the file **employees**. The keys are the third field (field 2) and the fifth field (field 4). Within the field 2 only, punctuation is ignored (the `-d` option).

```
sort +2d -3 +4 -5 employees
```

The next example is similar. The only difference is that instead of the first key ending just before field 3, the key ends just before the first character in field 3 that is not a **space** or **tab**.

```
sort +2d -3b +4 -5 employees
```
In this last example, field 2 is sorted with the punctuation ignored and in reverse order. Field 4 is sorted in the usual fashion.

```
sort +2dr -3 +4 -5 employees
```

**COMPARING TEXT FILES**

**Finding Repeated Lines—the uniq Program**

The `uniq` program eliminates repeated lines in a text file. The syntax is:

```
uniq [-c | -d | -u] [-field] [+position] [input [output]]
```

where `input` is the name of the input file and `output` is the name of the output file. The values of `field` and `position` are explained below.

`uniq` reads a file line by line and compares each line with the line before it. The output consists of each line that is not identical to the preceding line. If you do not specify an input file or an output file, `uniq` will use the standard input and standard output, respectively. Thus, you can use `uniq` as a filter.

Here is an example of how you might use `uniq`. Say that every time someone logs in, their userid is written to the end of a file called `users`. At a particular time, this file contains the following lines:

```
harley
harley
dianne
harley
michael
lisa
lisa
michael
```

Thus, four different userids have logged in a total of eight times. You are interested in those times when a user logs in with a userid different from the last one that was used. Enter the command:

```
uniq users
```

All the adjacent duplications will be eliminated and you will see:
If you are interested in how many times this happened, count the number of lines of output:

```
uniq | wc -l
```

*uniq* is often used with a sorted file. This ensures that all the duplications are adjacent; *uniq* will write one copy of each unique line. For example, consider the pipeline:

```
sort users | uniq
```

The output of the *sort* command is:

```
dianne
harley
harley
harley
lisa
lisa
michael
michael
```

*uniq* reduces this to:

```
dianne
harley
lisa
michael
```

Thus, we have a sorted list of the different userids that logged in. If you want to know how many different userids logged in, use:

```
sort users | uniq | wc -l
```

The `-c` option counts each set of lines. (`c` stands for “count.”) *uniq* will display a number in front of each line showing how many times that line was duplicated. For example,

```
uniq -c users
```

produces:
If you sort the file first, you will get the total number of times each line appears in the file.

```
sort users | uniq -c
```

produces:

```
1 dianne
3 harley
2 lisa
2 michael
```

The `-d` option writes only those lines that are duplicated. (d stands for "duplicate." ) For example,

```
sort users | uniq -d
```

will display only those userids that logged in more than once:

```
harley
lisa
michael
```

The `-u` command displays only those lines that are not duplicated. (u stands for "unique." ) For example,

```
sort users | uniq -u
```

will display those userids that logged in exactly once:

```
dianne
```

One idiosyncrasy of `uniq` is that you cannot use more than one option at a time. (Look carefully at the syntax.) In practice, this is not much of a limitation. It is not necessary to be able to combine the `-d` and `-u` options because this is exactly how `uniq` behaves with no options.

Like the `sort` program, `uniq` recognizes fields. However, instead of letting you specify which fields to process, all `uniq` allows you to do is skip over fields at the beginning of each line. Although the field specifications for `uniq` and `sort` are different, they are just similar enough to confuse you completely if you are not careful.

`uniq` assumes that the fields of each line are separated by `space` or
Comparing Text Files 251

\texttt{tab} characters. The fields are numbered starting from zero. (In other
words, the first field is field 0.) To start the comparison at a particular
field, use a \texttt{-} character followed by the number of the field.

For example, say that \texttt{accounts} is a text file in which each line has
several fields. The following \texttt{uniq} command starts comparing at the
beginning of the third field (field 2) of each line.

\begin{verbatim}
uniq -2 accounts
\end{verbatim}

To start comparing at a particular position within the field, use a \texttt{+}
character followed by that position. Positions are also numbered
starting from zero. For example, the following command starts com­
paring at the fourth position (position 3) of the second field (field 1).

\begin{verbatim}
uniq -1 +3 accounts
\end{verbatim}

These last two commands are equivalent. They both compare starting
from the fifth character in the line (field 0, position 4).

\begin{verbatim}
uniq -0 +4 accounts
uniq +4 accounts
\end{verbatim}

Finding Common Lines in Two Files—the \texttt{comm} Command

The \texttt{comm} command compares two files, looking for common lines.
\texttt{comm} stands for “common.”) The syntax is:

\begin{verbatim}
comm [-123] file1 file2 | - file2 | file1 -
\end{verbatim}

where \texttt{file1} and \texttt{file2} are the names of text files that you want to
compare.

\texttt{comm} reads both files, line by line. As it reads, \texttt{comm} writes three
columns of output. The first column contains the lines that are only in
\texttt{file1}. The second column contains the lines that are only in \texttt{file2}. The
third column contains the lines that are common to both files.

Here is an example: Say that \texttt{old.report} and \texttt{new.report} are two text
files that contain similar copies of a report. If you want to see how the
two reports differ, use:

\begin{verbatim}
comm old.report new.report
\end{verbatim}

(This can also be done with the \texttt{sdiff} command, explained in the next
section.)

\texttt{comm} is especially useful with sorted files. Here is an example.
\texttt{words1} and \texttt{words2} are two text files that contain many different words,
each on a separate line. You want to see which words are in which files.
First, sort the files and eliminate the duplicate lines. Put the sorted files into two temporary files, temp1 and temp2. There are two ways to do this:

```
sort -u names1 > temp1
sort -u names2 > temp2
```
or

```
sort names1 | uniq > temp1
sort names2 | uniq > temp2
```

Now, compare the two temporary files and look for common lines.

```
comm temp1 temp2
```
Finish by erasing the temporary files.

```
rm temp1 temp2
```

`comm` writes to the standard output. Thus, you can pipe the output to another program. For instance, in the last example you may want to make a printed report. Use:

```
comm temp1 temp2 | pr | lpr
```

Since `comm` needs two input files, it cannot act as a true filter. However, you can specify that one or the other of the files is to be read from the standard input. To do this, use a - character instead of a file name. (Look carefully at the syntax.)

Here is an example: You want to combine the text files `saturday` and `sunday` and compare the result with the text file `weekend`. Use:

```
cat saturday sunday | comm - weekend
```
In this case, the output of `cat` serves as the first file for `comm`. If you want the combined files to serve as the second file for `comm`, use:

```
cat saturday sunday | comm weekend -
```
`comm` will generate meaningless output if you specify two file names that are the same, or if you use two - characters.

The options are straightforward. -1 suppresses the output of the first column; -2 suppresses the output of the second column; and -3 suppresses the output of the third column.

For example, say that you want to find those lines that are in one or the other file but not in both files. Suppress the third column:

```
comm -3 file1 file2
```
You can combine the options. For example, to display only those lines that are in the first file, suppress the second and third columns:

```
comm -23 file1 file2
```

If you suppress all three columns:

```
comm -123 file1 file2
```

`comm` will display nothing.

Comparing Two Files Side by Side—the `sdiff` Command

The `sdiff` command produces a side-by-side display of two files, showing which lines are identical and which lines are different. (`sdiff` stands for “show differences.”) The syntax is:

```
sdiff [-ls] [-w width] [-o output] file1 file2
```

where `width` is a number; `output` is the name of a text file; and `file1` and `file2` are the names of the two text files that you want to compare.

`sdiff` produces two columns of output. The lines from the first file are on the left; the lines from the second file are on the right. When lines do not match, `sdiff` puts a special character between the columns. A `|` character indicates that the lines are partially different. A `<` character indicates that a line from the first file was not found in the second file. A `>` character indicates that a line from the second file was not found in the first file.

Here is an example. Say that `names1` and `names2` are two text files. `names1` contains:

```
harley
dianne
misty
lisa
```

and `names2` contains:

```
harley
diane
lisa
michael
```

If you enter:

```
sdiff names1 names2
```
you will see:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>harley</td>
<td>harley</td>
</tr>
<tr>
<td>dianne</td>
<td>diane</td>
</tr>
<tr>
<td>misty</td>
<td>&lt;</td>
</tr>
<tr>
<td>lisa</td>
<td>&gt; michael</td>
</tr>
</tbody>
</table>

The lines with **harley** and **lisa** are the same in both files. The lines with **dianne** and **diane** are partially different. The line with **misty** is only in the first file. The line with **michael** is only in the second file.

There are three options that you can use to control how the lines are displayed. The **-l** option only displays the left side when two lines are identical. (l stands for "left.") For example,

```
    sdiff -l names1 names2
```

produces:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>harley</td>
<td>harley</td>
</tr>
<tr>
<td>dianne</td>
<td>diane</td>
</tr>
<tr>
<td>misty</td>
<td>&lt;</td>
</tr>
<tr>
<td>lisa</td>
<td>&gt; michael</td>
</tr>
</tbody>
</table>

The **-s** option does not display identical lines. (s stands for "same.") For example,

```
    sdiff -s names1 names2
```

produces:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>dianne</td>
<td>diane</td>
</tr>
<tr>
<td>misty</td>
<td>&lt;</td>
</tr>
<tr>
<td></td>
<td>&gt; michael</td>
</tr>
</tbody>
</table>

**Warning:** Owing to a flaw in the program, **sdiff** will sometimes display strange lines when you use the **-s** option. These lines are really output from the **diff** program (described in the next section). Ignore them.

Left to itself, **sdiff** will produce lines of output that are 130 characters long. This is suitable for printers that use wide paper but not for much else. You can use the **-w** option to set the size of the output lines. (w stands for "width.") Use **-w** followed by the maximum number of positions that you want on each line of output.

For example, 80 is a good width for displaying information. The following command sets this width:

```
    sdiff -w 80 names1 names2
```
Note: If necessary, \texttt{sdiff} will leave off the ends of lines to make the lines fit within the required width.

As with most commands, you can redirect the output of \texttt{sdiff} to a text file. However, \texttt{sdiff} provides a more useful tool that allows you to merge lines selectively from the two files as they are compared. Use the \texttt{-o} option followed by the name of a text file. (\texttt{o} stands for "output.") When you use this option, the \texttt{-l} and \texttt{-s} options have no effect.

\texttt{sdiff} will send one copy of each identical line to the file you specify. When differences are found, \texttt{sdiff} displays them and then displays a \% character. The \% is a prompt. You can now enter a command to indicate what to do with these lines. I will describe the most useful of the commands that you can use.

If you want to send the left column to the output file, enter \texttt{l} (for "left.") If you want to send the right column, enter \texttt{r} (for "right.") If you enter \texttt{s} (for "silent"), \texttt{sdiff} will stop displaying identical lines (even though they will still be sent to the output file). If you enter \texttt{v} (for "verbose"), \texttt{sdiff} will start displaying identical lines. As soon as you enter a \texttt{l} or \texttt{r} command, \texttt{sdiff} will continue with the comparison.

The Command Reference Manual describes other commands that you can use to let you edit a line before you send it to the output file. You should be aware that the editor that is used is \texttt{ed}, not \texttt{vi}.

Note: \texttt{sdiff} uses the output from the \texttt{diff} program (described in the next section) to produce its own output. \texttt{diff} tends to group several different lines together. Thus, when you use the \texttt{-o} option, you may not be able to decide on each different line separately. You may be forced to decide on groups of different lines.

\section*{Comparing Two Text Files—the \texttt{diff} Command}

The \texttt{diff} program compares two files. (\texttt{diff} stands for "differences.") This is the command to use when you want to find out if two text files are the same. If you want to compare two binary files, use the \texttt{cmp} program. (See the Command Reference Manual for details.)

Omitting the less important options, the syntax for the \texttt{diff} command is:

\begin{verbatim}
  diff [-be] file1 file2  | - file2  | file1 -
\end{verbatim}

where \texttt{file1} and \texttt{file2} are the names of two text files that you want to compare.

The \texttt{diff} command is a curiosity in that, except for one particular case, the output is not important. \texttt{diff} will compare the two files you
specify. If the files are identical, **diff** will not display anything. If the two
files differ, **diff** will produce output that you can safely ignore.

Here is an example of a **diff** command that compares the two text
files **old.report** and **new.report**:

```bash
diff old.report new.report
```

Since **diff** requires two input files, it cannot act as a true filter. However,
as with the **comm** command, you can specify that one of the files is to be
read from the standard input by using a - character instead of a file
name.

Here are two examples: Each example combines the files **saturday**
and **sunday** and compares the result to the file **weekend**. The first
example reads the combination file as **file1**. The second example reads
the combination file as **file2**.

```bash
cat saturday sunday | diff - weekend
cat saturday sunday | diff weekend -
```

When the two files are different, **diff** generates output that resembles
editing commands. These commands tell you how you would have to
change the files to make them identical.

Here is the most useful way to compare text files: First, use **diff** to
see if the files are identical. If they are not and you care about the
differences, use **comm** or **sdiff**. Don't bother with the output of **diff**. (As a
matter of fact, the job of **sdiff** is to take the output of **diff** and produce
readable output.)

The -b option tells **diff** to ignore any space or tab characters at the
end of a line. As well, all groups of space or tab characters are to be
considered the same. For example, with the -b option, the following two
lines are considered identical even though there are more space char-
acters in the second line.

```bash
hello to you
hello      to      you
```

### Using **diff** to Maintain Versions of a Text File

If you use the -e option, **diff** will produce an editing script that will
recreate **file2** from **file1**. (e stands for "edit.") This editing script is
suitable for either the **ex** or **ed** editor.

Here is how you might use the -e option. Say that **data** is a very
large text file that you modify every now and then. You decide that you
want to keep the old versions of the file whenever you modify it.
However, **data** is so large that it would require a lot of storage to keep
more than one copy. This is especially wasteful if you only make small changes.

Whenever you change the file, you can use the **diff** command with the `-e` option to create an editing script that can recreate the previous version. Thus, you only need to store the instructions for recreating the old version from the new version.

Here is an example that you can use as a guideline. Before you actually do this with an important file, practice with an unimportant file to make sure that you understand how it all works.

You are about to make changes to the text file **data**. First, make a copy of the file and call it **data.1**:

```
cp data data.1
```

Now, change the file **data** as you wish. Once this is done, create an editing script, called **script.1**, that can recreate **data.1** from **data**:

```
diff -e data data.1 > script.1
```

You can now erase the old version of **data**:

```
rm data.1
```

When the time comes to recreate **data.1**, start by making a copy of **data** and call it **data.1**:

```
cp data data.1
```

You must now change **data.1** into the old version of **data**. Start the **ex** editor using the file **data.1**:

```
ex data.1
```

**ex** will display a colon (its prompt character) to let you know that you can enter a command. Use the **so** command to execute the instructions in **script.1**. (This command is explained in Chapter 13. Editing scripts are explained in Chapter 14.)

```
enter: so script.1
```

Now, write the editing buffer to the file **data.1** (use the **w** command) and stop **ex** (use the **q** command):

```
w
q
```

**data.1** is now restored.

If you modify a large file repeatedly, you can use this technique to create editing scripts for as many versions as you want. It is a good idea to number the scripts in order. For instance, in this example you would
call the next script \texttt{script.2}, and it would recreate \texttt{data.2} from \texttt{data}. If necessary, you could then use \texttt{script.1} to recreate \texttt{data.1} from \texttt{data.2}.

For example, if you follow this plan through four versions, you will end up with four scripts. If you ever need to recreate the original file, you can start \texttt{ex} and enter:

\begin{verbatim}
so script.4
so script.3
so script.2
so script.1
\end{verbatim}

Just make sure to use the scripts from newest to oldest.

If you plan to maintain documents or programs on a large scale, Xenix has a collection of programs that provide a complete maintenance system. It is called the Source Code Control System (SCCS), and it is part of the Software Development System. See the Software Development Guide for details.
CHAPTER 17

SHARING AND RESTRICTING ACCESS TO FILES

INTRODUCTION

PERMISSIONS

CONTROLLING THE ACCESS TO A FILE
INTRODUCTION

One of the most important features of the Xenix file system is that it allows you to control who uses your files. You can share your files or restrict access to them, as you see fit. In this chapter, you will learn how to understand and use this facility.

New Words in This Chapter

GROUP
a set of userids; each userid is assigned to a group by the system manager

PERMISSIONS
an attribute of a file that specifies how that file may be accessed

PERMISSIONS

What Are Permissions?

In Chapter 7, I explained that every file has an owner. The owner is the userid that controls access to the file. Usually, the owner is the userid that created the file.

Within Xenix, each userid belongs to a GROUP. The groups are set up by the system manager. Whenever a new user is registered, the manager specifies what group that userid will be in. A group is usually several people who are working on a project together or who need to share related files. For example, there might be a group named sales, consisting of the userids of all the salespersons in a company.

Xenix maintains a system of PERMISSIONS that specifies which userids are allowed to access a file. Each file has three sets of permissions. The first set specifies how the owner may access the file; the second set specifies how the members of the owner's group may access the file; and the third set specifies how everybody else may access the file.

You can change the permissions for your files any time you want. You can set permissions to share data or to restrict access to data. For example, say that you have a file that contains a mailing list. You may
Permissions set the permissions so that anyone may read the file, but only you or members of your group may make changes.

If you have a secret file, you can set the permissions so that only you may look at it. You can even specify that no one else may look in a directory to see the names of your files.

Permissions for Ordinary Files

You can specify permissions for ordinary files and for directories. The permissions consist of one or more of the values \( r \), \( w \), and \( x \). (These names are abbreviations for "read," "write," and "execute.")

For an ordinary file, the values mean:

- \( r \) a user may read the file
- \( w \) a user may write to the file
- \( x \) a user may execute the file

(This last value only makes sense for files that can be executed.)

You can display the permissions for an ordinary file by using the `ls` command with the `–l` option. Here is an example of the output of such a command:

```
-rwxr-xr-- 1 joe sales 9940 May 17 23:44 /usr/acme/report
```

The first character shows the type of file. In this case, the `-` character means the file is an ordinary file. The owner is `joe` and the group is `sales`.

The next nine characters show the permissions. Think of them as three sets of three:

- `rwx` for the owner
- `r-x` for the owner's group
- `r--` for everyone else

The first three characters show the permissions for the owner; the next three characters show the permissions for the owner's group; the last three characters show the permissions for everyone else. The permissions are listed in the order \( r \), \( w \), and \( x \). If a permission is not given, a `-` character is used.

In this example, the owner has the permissions \( r \), \( w \), and \( x \); the owner's group has the permissions \( r \) and \( x \); and everyone else has only the \( r \) permission. In other words, the owner can read, write, or execute the file; the owner's group can read and execute the file; and everyone else can only read the file.

Here is another example:

```
-r-------- 1 george managers 9940 May 17 23:44 /usr/acme/forecast
```
The permissions are:

```
-r--  ---  ---
```

In this example, the owner is `george` and the group is `managers`. For security reasons, the owner has set the permissions so that no one else can access the file. As well, the owner has only read permission. This makes sure that the owner can't accidentally change the file.

**Permissions for Directories**

As with an ordinary file, the permissions for directories are designated by `r`, `w`, and `x`. However, the meanings of these letters are different:

- **r**: a user may display the names of files in the directory
- **w**: a user may create and remove files within the directory
- **x**: a user may use the files in the directory

The `w` permission allows a user to modify the directory. In particular, the user can create ordinary files and sub-directories. However, he or she cannot do so without also having the `r` and `x` permissions. The `x` permission allows a user to search the directory and to use the `cd` command to make the directory his or her working directory.

You can display the permissions for a directory by using the `ls` command with the `-ld` options. Here is an example of the output of such a command:

```
drwxr-xr-x   1   joe  sales  9940 May 17 23:44 /usr/acme/summaries
```

The first character shows the type of file. In this case, the `d` character means the file is a directory. The owner is `joe` and the group is `sales`. The permissions are:

```
 rwx  r-x  r-x
```

The owner has full access to the files in the directory. The members of the group and everyone else can use the directory, but they can't create or remove files.

Here is another example:

```
drwx------   1   george  managers  9940 May 17 23:44 /usr/acme/secrets
```

The permissions are:
CONTROLLING THE ACCESS TO A FILE

Changing Permissions—the chmod Command

When you create an ordinary file Xenix assigns the permissions:

```
rwx  r--  r--
```

This means that everyone can read the file, but only the owner can write to the file.

When you create a directory, Xenix assigns the permissions:

```
rwx  r-x  r-x
```

This means that everyone can use the directory, but only the owner can create and remove files.

If you want to change the permissions for a file, use the `chmod` command. (`chmod` stands for "change mode.") A regular user can change permissions only for the files for which he or she is the owner. The superuser can change permissions for anybody's files.

The syntax for this command is:

```
chmod  mode file...
```

where `file` is the name of an ordinary file or directory whose permissions you want to change.

`mode` is a three-digit number that represents the permissions that you want to assign to the file. The first digit represents the owner's permissions; the second digit represents the group's permissions; and the third digit represents the permissions for everyone else.

For example, consider the command:

```
chmod 744 test reports
```

This command changes the permissions for the files `test` and `reports`.

The three digits represent the three sets of permissions.

The meaning of the digits is as follows:
Sharing and Restricting Access to Files

0 no permissions
1 x
2 w
3 w and x
4 r
5 r and x
6 r and w
7 r, w, and x

Another way to look at it is that r has the value 4, w has the value 2, and x has the value 1. To get a combination, add the values of the permissions you want. For example, for r and x the value is 4 + 1 or 5.

Here are some examples: Say that report and forecast are text files. You want to change the permissions of report so that the owner and his or her group may read and write to the file and no one else can do anything. The owner and the group should have r and w permissions, while everyone else has none. Use:

chmod 660 report

You want to change forecast so that the owner can read the file and no one else can access it. Use:

chmod 400 forecast

To give everyone full permissions for report, use:

chmod 777 report

Here is an example in which you want to add one permission. You have just created the file setup. You display the permissions using the \texttt{lsof -l} command and you see:

\texttt{-rw-r--r-- 1 harley expert 9940 May 17 23:44 setup}

Thus, the permissions are:

\texttt{rwx r-x r-x}

corresponding to 644. You want to give everyone permission to execute the file. (This is what you would do with a shell script as described in Chapter 20.) The new permissions should be:

\texttt{rwx r-x r-x}

The digits that correspond to this are 7, 5, and 5. Use the command:

chmod 755 setup
Changing Owners and Groups—the `chown` and `chgrp` Command

Xenix has commands that allow you to change the owner and the group for a file. This may come up if you change your userid or your group. To change the owner of a file, use the `chown` command. (`chown` stands for "change owner.") The syntax is:

```
chown owner file...
```

where `owner` is the userid of the new owner, and `file` is the name of the file for which you want to change the owner.

For example, to change the owner of the file `report` to the userid `george`, use:

```
chown george report
```

To change the group of a file, use the `chgrp` command. (`chgrp` stands for "change group.") The syntax is:

```
chgrp group file...
```

where `group` is the name of the new group, and `file` is the name of the file for which you want to change the group.

For example, to change the group of the file `report` to the name `managers`, use:

```
chgrp managers report
```

Only the owner of a file or the superuser can change the owner and group for that file. (Of course, if you change the owner, you will lose control over the file.)

To see a list of all the userids, display the file `/etc/passwd` (note the spelling). Enter:

```
more /etc/passwd
```

This is the password file. (The actual passwords are encrypted.) Each line represents one userid and contains several entries separated by colons. The first entry in each line is the userid.

To see a list of all the groups, display the file `/etc/group`.

```
enter: more /etc/group
```

Each line represents one group and contains several entries separated by colons. The first entry in each line is the name of the group.
CHAPTER 18

THE XENIX MAIL SYSTEM

INTRODUCTION
UNDERSTANDING THE MAIL SYSTEM
RECEIVING MAIL
DISPOSING OF MAIL
SENDING MAIL
REVIEW
INTRODUCTION

Xenix provides several ways to send information between users. You can use the write program to communicate with someone who is logged in. You can set file permissions to share information with another user. In this chapter, you will learn how to use the mail system to send and receive messages in the form of text files.

New Words in This Chapter

MESSAGE
a text file that is sent from one userid to another by the Xenix mail system

HEADING
the first part of a message, which contains general information about the message

BODY
the second part of a message, which contains the text of the message

BLIND COPY
a copy of a message sent in such a way that only the sender and recipient know who received the copy

HEADER
a one-line description of a message

COMMAND MODE
a state in which a user can enter commands to the mail program

CURRENT MESSAGE
the last message that was displayed

MESSAGE LIST
a specification of one or more messages

COMPOSE MODE
a state in which a user can type a message from within the mail program
UNDERSTANDING THE MAIL SYSTEM

Overview

Xenix provides a system that allows you to send and receive MESSAGES in the form of text files. You can send messages to another userid, to a group of userids, and even to yourself. A message can be anything that can be stored in a text file; a memo, a letter, a program, and so on. Here is a brief outline of how the message system works.

Xenix maintains a set of files called system mailboxes in the directory `/usr/spool/mail`. Each userid has its own system mailbox. The name of your system mailbox is the same as your userid. For example, if your userid is `david`, your system mailbox is `/usr/spool/mail/david`.

To send or receive a message, you use the `mail` program. Whenever anyone sends you a message, Xenix adds it to the end of your system mailbox. If you are logged in at the time, you will receive the notice:

```
you have mail
```

If you are editing, you will not see the notice until you stop the editor. Every time you log in, Xenix checks your system mailbox and lets you know if you have messages waiting.

When you have messages waiting, you use `mail` to read them. After you read a message, you can either save it or delete it. If you want, you can print the message or forward it to another userid.

You also use the `mail` program to send messages. `mail` will copy the message to the end of the recipient's system mailbox. When you send a message you can send a copy to other userids or to yourself. You can even have `mail` let you know that your message was delivered properly.

You can receive mail not only from other userids but from the Xenix system itself. Here are some examples:

- When you log in for the first time, there will be a message from Xenix welcoming you to the system.
- If you are interrupted while editing, `vi` will send you a message telling you how to recover your file.
- If you print a file by using `lpr` with the `-m` option, you will receive a message when the printing is complete.

The Xenix message system is a complex set of programs developed at U.C. Berkeley. It consists not only of the `mail` program but also a daemon called `mailer` that ties everything together. Like other Berkeley programs, `mail` will do far more than you usually need. In this chapter, I
will describe the most important aspects of mail. For an exhaustive reference, see both the Command Reference Manual and the Basic Operations Guide.

The Parts of a Message

Every message has two parts: a HEADING and a BODY. The heading comes first and contains general information about the message. The body contains the text of the message.

Here is an example of a message:

```
From alfred Thu Aug 15 12:54:02 1985
To: edna
Subject: meeting next Monday
Cc: stewart sam ruth jessica
Return-receipt-to: alfred
Date: Thu Aug 15 12:54:01 1985

Remember, our regular Thursday afternoon meeting is re-scheduled for next Monday. If you have anything to add to the agenda, please send me a message by Friday morning.
```

The heading is the first six lines; the body is the last three lines. Here are the meanings of the lines in the heading:

<table>
<thead>
<tr>
<th>Line</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>line 1</td>
<td>the userid of the sender followed by the date and time the message was received</td>
</tr>
<tr>
<td>line 2</td>
<td>the userid of the recipient</td>
</tr>
<tr>
<td>line 3</td>
<td>the subject of the message</td>
</tr>
<tr>
<td>line 4</td>
<td>a list of userids that are to receive copies of the message</td>
</tr>
<tr>
<td>line 5</td>
<td>a list of userids that are to receive an automatic acknowledgement that the message has been sent</td>
</tr>
<tr>
<td>line 6</td>
<td>the date and time the message was sent</td>
</tr>
</tbody>
</table>

The abbreviation **Cc** stands for “carbon copy.” It is an old term, often used in business letters, to let the recipient know that someone has been sent a copy of the letter. When you send a message, the **Cc** line is included as part of the heading.

You can also include a line in the heading that consists of **Bcc** followed by a list of userids. **Bcc** stands for BLIND COPY. This means that the specified userids are to receive a copy of the message, but that no one else is to know about it. To maintain secrecy, the **Bcc** line is not included as part of the message.
The only parts of a message that are always included are the first two lines of the heading (showing the sender and the recipient) and the date. All the other parts, including the body of the message, are optional.

RECEIVING MAIL

Starting the Mail Program

The `mail` command has several forms. To read your messages, enter the `mail` command with no arguments:

```
mall
```

If your system mailbox is empty, `mail` will display:

```
No messages.
```

and stop. If not, `mail` will display something like this:

```
2 messages:
  2 ruth Fri Aug 16 16:30 7/184 "refreshments"
  1 alfred Thu Aug 15 12:48 9/339 "meeting next Monday"
```

The first line tells you what version of the `mail` program you are using and the date and time. It also indicates that you can enter `?` for a list of all the `mail` commands. The second line tells how many messages are in your system mailbox.

Following the first two lines are a list of one-line descriptions of each message. These descriptions are called HEADERS. Do not confuse a header (a one-line summary) with a heading (the first part of a message).

The header tells you the number of the message, the userid of the sender, and the date and time that the message was sent. The last two parts of the header show the number of lines and characters in the message and the subject. Headers are listed from newest to oldest.

The first header in the example above describes message number 2. It was sent by `ruth` and is 7 lines (184 characters) long. The subject is "refreshments."
Using \texttt{mail} Commands

As soon as you see the list of headers, you are in COMMAND MODE. This means that you can enter commands to display and dispose of your messages. At any time, you can display a list of all the \texttt{mail} commands by entering:

\texttt{?}

If you want to enter a Xenix command while you are working with \texttt{mail}, enter \texttt{l} followed by the command. For example, to display the time, enter: \texttt{idate}

If you want to enter a series of Xenix commands, enter: \texttt{shell}

\texttt{mail} will start a new copy of the shell. When you want to stop the shell and return to \texttt{mail}, press \texttt{<CTRL-d>} to send the \texttt{eof} (end of file) signal.

If you are using \texttt{mail} and someone sends you a message, you will be told to enter:

\texttt{restart}

to access the new message.

When you are finished, stop the \texttt{mail} program by using the \texttt{q} command. (\texttt{q} stands for “quit.”) The syntax is:

\texttt{q}

Displaying Messages

The easiest way to display your messages is to press <ENTER> again. You can keep on pressing <ENTER> to work your way through all the messages.

At any time, the last message that you displayed is called the CURRENT MESSAGE. In general, pressing <ENTER> will display the message that comes after the current message. If there are no more messages, \texttt{mail} will display:

\texttt{Can't go beyond last message}

If you want to display a particular message, enter its number. For example, if you want to display message 8,

enter: 8
This message now becomes the current message.

To display more than one message at a time, you can use the \texttt{p} command. (\texttt{p} stands for "print.") Remember, the Unix tradition is to use "print" to mean "display.")

The syntax for the \texttt{p} command is:

\begin{Verbatim}
\texttt{p [list]}
\end{Verbatim}

where \texttt{list} is a MESSAGE LIST.

Message lists are used with many of the \texttt{mail} commands to refer to a group of messages. When you use a message list, the command acts on each message you specify. If you do not use a message list, the command acts on the current message.

The simplest form of a message list is several numbers separated by \texttt{space} characters. For example, to display messages 4, 5, and 6,

\begin{Verbatim}
enter: \texttt{p 4 5 6}
\end{Verbatim}

You can indicate a range of messages by using two numbers separated by a hyphen. For example, the following command is equivalent to the last one:

\begin{Verbatim}
\texttt{p 4-6}
\end{Verbatim}

A list can also contain userids. For example, the command:

\begin{Verbatim}
\texttt{p sam}
\end{Verbatim}

displays all the messages from userid \texttt{sam}. If you combine numbers and userids in the same list, it refers to the messages with those numbers that are from those userids. For example, to display all the messages from 10 to 20 that are from \texttt{sam}, use:

\begin{Verbatim}
\texttt{p 10-20 sam}
\end{Verbatim}

Remember, if you do not specify a message list, most commands will act on the last message that was displayed. For example, if you enter:

\begin{Verbatim}
\texttt{p}
\end{Verbatim}

\texttt{mail} will redisplay the current message.

If you have some long messages, you can use the \texttt{t} command to display the first five lines of each one. (\texttt{t} stands for "top.") The syntax is:

\begin{Verbatim}
\texttt{t [/list]}
\end{Verbatim}

where \texttt{list} is a message list. For example, to display the first five lines of messages 3, 5, 6, 7, 8, and 9,

\begin{Verbatim}
enter: \texttt{t 3 5-9}
\end{Verbatim}
The `si` command displays the number of characters in a message. (`si` stands for "size.") The syntax is:

```
  si [list]
```

where `list` is a message list. For example, to display the number of characters in message 6,

```
Enter:   si 6
```

### Displaying Headers

There are two commands that you can use to display headers. The first is the `h` command. (`h` stands for "header.") The syntax is:

```
  h [ + | - | list ]
```

where `list` is a message list.

With no arguments, the `h` command displays the headers for the messages in your system mailbox. For example:

```
Enter:   h
```

If you have many messages, `mail` will display 18 headers at a time. To display the next group of 18 headers,

```
Enter:   h +
```

To display the previous group,

```
Enter:   h -
```

You can also display the headers for particular messages by specifying a message list. For example, to display the headers for all the messages from userid `sam`,

```
Enter:   h sam
```

The `string` command will display the headers of all the messages that contain a specified string of characters. The syntax is:

```
  string string [list]
```

where `string` is a string of characters and `list` is a message list. For example, say that you want to display a message that `sam` sent you, in which he talked about the "farkle plus" project. You can't remember which message it is.

```
Enter:   string 'farkle plus' sam
```
This command searches all the messages from userid sam. Notice that this particular character string needs to be enclosed in single-quotes because it contains a space character.

DISPOSING OF MAIL

Removing Messages

There are several ways that you can dispose of a message. You can save it, print it, forward it to another user, or remove it from the system mailbox.

To remove a message, use the d command. (d stands for “delete.”) The syntax is:

\textbf{d \[list\]}

where \textit{list} is a message list. For example, to remove messages 3, 4, 5, 10, and 13,

\textit{enter: d 3-5 10 13}

To remove the current message,

\textit{enter: d}

\texttt{mail} does not actually erase a message until you stop the program. If you make a mistake, there are two ways that you can get a message back. First, you can use the \texttt{u} command. (\texttt{u} stands for “undelete.”) The syntax is:

\textbf{u \[list\]}

where \textit{list} is a message list. For example, to get back message 6,

\textit{enter: u 6}

To get back the current message,

\textit{enter: u}

The second way to get back messages is to stop the \texttt{mail} program by using the \texttt{x} command instead of the \texttt{q} command. (\texttt{x} stands for “exit.”) The syntax is:

\texttt{x}
When you stop by using `q`, `mail` erases all the messages that you removed. When you stop by using `x`, `mail` does not modify your system mailbox. No messages are erased.

**Saving, Printing, and Forwarding Messages**

The most useful way to save a message is to copy it to a file. To do this, use the `s` and `w` commands. (`s` stands for “save”; `w` stands for “write.”) The syntax for these commands is:

```
s [list] file
w [list] file
```

where `list` is a message list, and `file` is the name of a text file.

These commands copy the specified messages to the end of a text file. If you don’t specify a message list, the commands will copy the current message.

The difference between these two commands is that `s` copies the entire message, while `w` copies only the body of the message. For example, to copy the current message to the end of the text file `reports`, enter:

```
s reports
```

To copy the body of the current message, enter:

```
w reports
```

When you stop `mail` by using the `q` command, all the messages that you have copied are automatically erased. If you stop by using the `x` command, the messages are retained.

To print a message, use the `l` command. (`l` stands for “line printer.”) The syntax is:

```
l [list]
```

where `list` is a message list. For example, to print the current message enter:

```
l 5
```

When you print a message in this way, it is broken into pages by the `pr` command and sent to the print queue by the `lpr` command. (These commands are explained in Chapter 15.) Printing a message does not remove it from your system mailbox.

Aside from saving and printing a message, you can forward it (send a copy) to another user. Use the `f` command. (`f` stands for “forward.”)
The syntax is:

    f [number] userid...

where number is the number of a message, and userid is a userid.

`mall` will send a copy of the message you specify to the userids you specify. For example, to send a copy of message 5 to the userids 「curly」, 「larry」, and 「moe」,

```
    enter: f 5 curly larry moe
```

If you do not specify a message number, `mall` will forward the current message.

Forwarding a message does not remove it from your system mailbox. For example, say that you have just displayed a message. You decide to forward it to 「sam」 and then remove it.

```
    enter: f sam
    enter: d
```

## SENDING MAIL

### Sending an Existing Text File

To send a message to another user, use the `mall` command with the following syntax:

```
    mall [-b userid...] [-c userid...] [-r userid...] [-s string] recipient...
```

where userid is a userid; string is a character string; and recipient is the userid to which you want to send the message.

In the simplest case, you can use `mall` to send a copy of a text file that you have already created using the editor. In this case, redirect the standard input to the file. For example, to send a copy of the file 「report」 to userid 「sam」,

```
    enter: mall sam < report
```

If you want to send the same message to 「sam」, 「edna」, and 「alfred」,

```
    enter: mall sam edna alfred < report
```

Be careful that you don't accidentally type a `>` character instead of `<`. For example, if you enter:

```
    mall sam > report
```
it redirects the standard output instead of the standard input. The contents of report will be lost.

The options allow you to specify the values of the different lines of the heading as follows:

- **-b** list of userids to receive blind copies (Bcc)
- **-c** list of userids to receive copies (Cc)
- **-r** list of userids to receive a return receipt
- **-s** subject of message

Whatever follows the option becomes the value of that line in the heading. If this value contains space characters, enclose it in single-quotes.

For example, say that the text file report contains a message that you want to send to userid sam. The subject of the message is “The year-end report.” You want to send a copy of the message to alfred and a blind copy (that no one else should know about) to george. A return receipt is to be sent to david. Enter the command:

```
mail -b george -c alfred -r david -s 'Annual report' sam < report
```

Here is a simpler example: You want to send a message to sam about the new meeting schedule. The message is in the file meeting.

```
enter: mail -s 'New meeting schedule' sam < meeting
```

This last example sends the same message to sam with copies to george and alfred:

```
mail -c 'george alfred' sam < meeting
```

**Composing a Message from within mail**

You can use the mail command to send a message that already exists as a text file. However, you can also compose the message from within mail. To do this, simply enter the mail command with no redirection. For example, to send a message to userid sam,

```
Enter: mail sam
```

mail will ask you for the subject of the message. Once you enter the subject, mail will change to COMPOSE MODE. This means that whatever you type will become the body of the message. When you are finished typing, press <CTRL-d>. mail will display:

```
(end of message)
```
and send off the message.

Here is an example of composing a letter within `mail`. You begin by entering:

```
mail sam
```

The `mail` program starts and displays:

```
subject:
```

You enter:

```
next Monday's meeting
```

You are now in compose mode, so you type the body of the message:

```
Sam, I'm sorry that I can't make the meeting on Monday. My armadillo club is having its annual luncheon at the same time.
```

You now press `<CTRL-d>` to show that you are finished. `mail` displays:

```
(end of message)
```

and sends the message to userid `sam`.

When you are typing in compose mode, you can correct mistakes using `<BACKSPACE>` to erase the last character and `<CTRL-u>` to erase the current line. If you decide that you want to forget about the message completely, press `<DEL>` twice in a row.

In general, `mail` is reluctant to throw away messages without your approval. If a message is not sent for some reason, `mail` will add it on to the end of a file called `dead.letter` in your home directory. This will happen if you press `<DEL> <DEL>` in compose mode or if you try to send a message to a userid that does not exist.

### Changing from Command Mode to Compose Mode

Sometimes you will be reading your mail when you decide that you want to send a message. There are two ways that you can do this. In other words, there are two ways to change from command mode to compose mode.

If you want to reply to a particular message, use the `r` or `R` command. (`r` stands for “reply.”) The syntax is:

```
r [list]
R [list]
```
This command will put you in compose mode. Type your message as usual. When you are finished, press <CTRL-d>. `mail` will take care of the details of sending the message to the proper userid.

The difference between the two commands is that `r` will send a message only to the userid that sent you the original message. `R` will also send copies to all the userids in the `Cc` list of the original message.

As with most of the other `mail` commands, if you do not specify a message list, `r` and `R` will reply to the current message.

For example, let's say that you have just displayed a message from `sam`. You decide to reply to it and then remove it. You want to reply only to `sam`; not to anyone else in the `Cc` list.

```plaintext
enter:  r
```

You are now in compose mode. Type your message. When you are finished, press <CTRL-d>. `mail` will send the message to `sam`.

### Commands from within Compose Mode

As you are typing in compose mode, there are certain commands that you can use, all of which begin with a tilde ( ~) character. In this section, I will describe the most useful of these commands.

The `~h` command allows you to set and modify the values in the header. After you enter `~h`, `mail` will display each line of the header: To, Subject, Cc, Bcc, and Return-receipt. You can modify or ignore each line. To modify a line, use `<BACKSPACE>`. To ignore a line, press `<ENTER>`.

The `~v` command will start the `vi` editor. The body of the message that you are typing will be in the editing buffer. When you stop the editor, you will be back in `mail`, still in compose mode. This is helpful if you find the rudimentary facilities of compose mode inadequate for editing your message. Remember, once you are back in `mail`, you must press <CTRL-d> to send the message.

The `~p` command will display the message that you are currently typing. (`p` stands for "print," which really means "display.") This is useful if you want to check the message before you send it. If you want to make changes, you can start `vi` by using the `~v` command.

To execute a Xenix command from compose mode, enter `~!` followed by the command. For example, to display the time,

```plaintext
enter:  ~!date
```
Sending Messages to a Group

There are several ways that you can send a message to more than one userid. Here are some examples of the `mail` command, all of which send the same message to userids `sam`, `edna`, and `alfred`:

- `mail sam edna alfred < report`
- `mail -c 'edna alfred' sam < report`
- `mail -c edna -b alfred sam < report`
- `mail sam edna alfred`

The first three commands send the message in the file `report`. The first command sends a separate message to each userid. The second command sends the message to `sam` and sends copies (marked as such) to `edna` and `alfred`. The third message sends a copy to `edna` and a blind copy (that no one knows about) to `alfred`. The last command is similar to the first, except that you compose the message from within `mail`.

If you often send messages to the same group of people, `mail` gives you a way to refer to the group by a name. For example, say that the group `sales` consists of `sam`, `edna`, and `alfred`. If you send mail to `sales`, it will automatically go to each member of the group.

To define a group, use the `a` command. (`a` stands for "alias.") The syntax is:

```
a [group [userid...] ]
```

where `group` is the name of a group and `userid` is a userid. You can use any group name you want, as long as it is not a userid.

Here is an example of a command that defines the group `sales` to consist of the userids `sam`, `edna`, and `alfred`:

```
a sales sam edna alfred
```

If you want to see a list of all the groups that are currently defined, along with their members, use the `a` command with no arguments:

```
a
```

To see the members of a particular group, use the `a` command with the name of the group. For example, to check which userids are in the group `sales`,

```
enter:  a sales
```

Once a group is defined, you can use its name anywhere you would use a userid. For example, if you put the name `sales` in the `Cc` field of a message, a copy will be sent to every member of the group.
If you define a group by entering an `a` command from command mode, the definition only lasts until you stop the `mail` program. Usually, it is more convenient to define groups that you can use every time you send messages.

To do this, create a file called `.mailrc` in your home directory. Every time the `mail` program starts, it looks for this file. If the file exists, `mail` will execute all the commands in it.

Here is an example: Say that your userid is `david` and you want to define permanently the following groups:

- **sales**: sam, edna, alfred
- **managers**: bart, george
- **friends**: sue, sam, arnold

Use the editor and create the file `/usr/david/.mailrc`, with the following lines:

```
a sales sam edna alfred
a managers bart george
a friends sue sam arnold
```

Notice that the same userid can appear in more than one group (in this case, the userid `sam`).

You can now send messages to any one of these groups. For example, say that you want to send the message in the file `memo` to userid `sam`, with a blind copy to `bart` and `george`. The following two commands are equivalent:

```
mall -b 'bart george' sam < memo
mail -b managers sam < memo
```

**REVIEW**

**General Comments**

There are two concepts about the `mail` program that you should understand. First, you have an obligation to remove your messages from the system mailbox. It is too easy to let mail accumulate and use up space on the fixed disk.

Make a point of disposing of each message as soon as you read it. If you want to save it, don't leave it in the system mailbox. Use the `s` or `w` commands to copy the message to one of your own files.

Some people organize their messages according to topic or sender.
For example, you might make a directory called messages to hold those messages that you want to save. Within this directory, each file holds one set of messages.

For instance, you might keep every message that has to do with accounting in the file messages/accounting. Or you might save every message from userid bart in the file messages/bart. Whenever you want to save a message, copy it to the appropriate file.

For example, if you have just read a message from bart that you want to save,

```bash
enter: s messages/bart
```

(Remember, the s and w commands add messages to the end of a file.)

Each message you save takes up space. Be selective. If you need a permanent copy of a message, print it. As a general rule, when in doubt, throw it out.

The second concept that you must appreciate is that you cannot expect privacy. If you receive a message, you have no idea if anyone else has received a blind copy. If you send a message, there is nothing to prevent the recipient from forwarding it to someone else. You will find that some people are habitual forwarders—they will pass on anything.

Even your system mailbox is not secure. Another user can read your messages by using the mail command with the -u option. (See the Command Reference Manual for details.)

Be forewarned: The Xenix message system is convenient but it is not secure. Do not send messages that contain information that is embarrassing or secret (for example, love letters).

Summary

Here is a summary of all the mail commands that are described in this chapter. The following arguments are used:

- list is a message list
- number is the number of a message
- string is a character string
- file is the name of a text file
- command is a Xenix command
- recipient is a userid to whom you want to send a message
- userid is a userid
To read your mail,

enter: `mail`

To display messages, use the following commands:

- `<ENTER>`: display next message
- `number`: display message `number`
- `p [list]`: display message
- `t [list]`: display first five lines of message
- `si [list]`: display size of message
- `h [ + | - | list]`: display headers
- `string string [list]`: search for messages containing `string`

To process a message, use one of the following commands:

- `d [list]`: delete message
- `u [list]`: get back deleted message
- `s [list] file`: copy entire message to `file`
- `w [list] file`: copy body of message to `file`
- `l [list]`: print message
- `f [number] userid...`: forward message
- `r [list]`: reply to message
- `R [list]`: reply to message with copies

If you do not specify `list` or `number`, the command will act on the current message (the last one you displayed). While in command mode, you can use the following commands:

- `?`: display list of commands
- `l command`: execute Xenix command
- `shell`: start a copy of the shell
- `restart`: read newly received mail

To stop the `mail` program, use:

- `q`: stop mail
- `x`: stop mail; do not remove messages

To send a message from a file, use:

```
mail recipient <file
```

To send a message that you create from `mail`, use:

```
mail recipient
```

When you use the `mail` command to send a message, the following options specify lines in the heading:
-b list of userids to receive blind copies (Bcc)
-c list of userids to receive copies (Cc)
-r list of userids to receive a return receipt
-s subject of message

While in compose mode, you can use the following commands:

~h set and modify the values in the header
~v start the vi editor
~p display the message
~lcommand execute Xenix command
<CTRL-d> finished; send message

To define groups of userids, use:

a [group [userid... ] ]

To define permanent groups, put a commands in the file .mallrc in your home directory.
CHAPTER 19

USING DISKETTES FOR BACKUPS AND WITH DOS

INTRODUCTION

BACKUPS

A PERSONAL BACKUP SYSTEM

USING DOS FILES
INTRODUCTION

In Chapter 15, you learned how to make a file system on a diskette. In this chapter, you will learn how to use diskettes in two other ways: to make copies of important information (backups); and to share information with the Disk Operating System (DOS).

New Words in This Chapter

BACKUP
a copy of information

BACK UP
to make a copy of information

RESTORE
to recover lost information by copying a backup

PERIODIC BACKUP
(sysadmin) a backup of a file system that contains all the files

DAILY BACKUP
(sysadmin) a backup of a file system that contains only those files that have been added or modified since the previous periodic backup

BACKUPS

What Are Backups?

A BACKUP is a copy of information. When you work with information stored on disks, it is important to make a backup. This is the only way you have of replacing the information if the disk should go bad or if you should accidentally erase a file.

When you make a copy of information for safekeeping, we say that you BACK UP the information. Thus, you may back up a file, a directory of files, or even an entire file system. If you lose some information and you replace it by copying a backup, we say that you RESTORE the file, directory, or file system.

One of the jobs of the system manager is to maintain a backup of the entire Xenix system. If a user accidentally loses a file, he or she should
be able to count on the system manager having saved a copy of the file. (Although the copy may be a few days old.) Rarely, something will go wrong with the fixed disk and all of the files may be lost. In this case, the system manager must be able to restore the entire system.

On a smaller scale, a user who works with important information may wish to make a backup of one or more of his or her own files. This allows the user to be doubly sure that the information is safe.

Most of the time, Xenix users keep their files on the fixed disk. For safekeeping, backups are usually made using diskettes. Since diskettes are removable, they can be kept in a secure place. If information is especially important, you may wish to make copies on two separate diskettes.

There are several ways that you can make backups using diskettes. First, you can make a file system on a diskette and use it to store files in the regular manner. (This is described in Chapter 15.) You can copy your important files to the diskette. If you accidentally erase a file, all you need to do is mount the diskette and copy the backup file to one of your directories.

This method is fine for backing up a limited number of files. However, it may prove awkward if you need to systematically make regular backups of large amounts of information. To help you, Xenix comes with two programs specifically designed to maintain backup systems.

The **sysadmin** program allows you to maintain backups of entire file systems. This program can only be used by the superuser. It was designed to allow the system manager to maintain the integrity of a large system with many users.

The **tar** program is useful for backing up separate files or directories of files. Any user can use this program to make a personal backup system.

**Backing Up Individual Files**

The easiest way to back up individual files is to create a file system on a diskette (see Chapter 15). Whenever you want to back up a file, you can mount the file system and copy the file to the diskette using the **cp** command. (Of course, you will have to have access to the diskette drives.) Here is an example of how you might do this starting with a blank diskette.

First, you must prepare your backup system. Format the diskette and make a file system on it. Next, make a directory in which you will
mount the file system. For example, if your userid is **harley**, you might want to make the directory `/usr/harley/backup`. You only need to do these steps once.

Each time before you start work, mount the file system using this directory. For example,

```
/etc/mount /dev/fd0 /usr/harley/backup
```

Leave the file system mounted as you work. Whenever you are modifying an important file, stop every half hour and copy the file to the diskette. For example, say that your working directory is `/usr/harley/book`. To copy the file `chapter.1` to the diskette, use either of these commands:

```
cp chapter.1 /usr/harley/backup
```

```
cp chapter.1 ./backup
```

When you are finished working, dismount the file system using:

```
/etc/unmount /dev/fd0
```

Remove the diskette and store it in a safe place. Take care to label the diskette appropriately. (For example, “Harley—BACKUP”.)

If you ever need to restore a file, mount the diskette and copy the file to one of your regular directories.

The question arises, How often should you stop work and make a backup? The answer, Decide how much work you are willing to lose and stop that often. For example, if you are willing to lose up to a half hour’s work, do a backup every half hour.

As you work, it is convenient to leave the backup diskette mounted at all times. This may be impractical. (For instance, you may have to share the diskette drives.) If so, mount your diskette just before you do the backup.

**Backing Up File Systems—the sysadmin Program**

**sysadmin** stands for “system administration.” It is a program that was designed to allow the system manager to maintain backups of entire file systems.

Before you can use the program, you must have some formatted diskettes available. (Formatting is explained in Chapter 15.) You do not need to make file systems on the diskettes.

**sysadmin** will only use high-capacity diskettes. The number that you need depends on how many files are stored on your fixed disk. Large systems may require as many as nine high-capacity diskettes.
As I explained in Chapter 15, Xenix consists of two file systems, root and usr. These file systems reside on the fixed disk and are mounted every time you start Xenix. The sysadmin program allows you to back up each file system separately.

The usr file system contains all the files under the /usr directory. This includes many of the programs that interpret the Xenix commands, as well as all the files that belong to the users. The root file system contains the files that are not under /usr. These are the files under the directories /bin, /dev, /etc, /lib, /lost+found, /mnt, and /tmp; as well as any files in the root / directory.

In order to be able to restore all of Xenix, you must backup both of these file systems. It is especially important to back up the usr file system as it contains everybody's work.

To back up a file system, log in as the superuser (userid root) and enter the sysadmin command. The syntax is:

```
sysadmin
```

You will see the following menu (list of choices):

```
<table>
<thead>
<tr>
<th>Type</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>to do daily backup (root file system)</td>
</tr>
<tr>
<td>2</td>
<td>to do daily backup (usr file system)</td>
</tr>
<tr>
<td>3</td>
<td>to do a periodic backup (root file system)</td>
</tr>
<tr>
<td>4</td>
<td>to do a periodic backup (usr file system)</td>
</tr>
<tr>
<td>5</td>
<td>to get a backup listing</td>
</tr>
<tr>
<td>6</td>
<td>to restore file(s)</td>
</tr>
<tr>
<td>q</td>
<td>to quit</td>
</tr>
</tbody>
</table>
```

sysadmin will then ask you to choose a particular task. Enter the number for the first task that you want. When this is finished, enter the number for the next task. When you are done with the program, enter q to quit.

As you can see, sysadmin performs two kinds of backups. A periodic backup backs up the entire file system. A daily backup backs up only those files that have been modified since that last periodic backup.

If you are the system manager, it is important to establish a regular routine for performing the backups. Here is a routine that many people find satisfactory.

At the beginning of each week (say, Monday morning), do a periodic backup on both file systems. On the other days of the week (say, each morning), perform a daily backup on both file systems. As you do a backup, sysadmin will tell you when to insert a new diskette. Take care
to label your diskettes so you don’t get them mixed up. (Say, “backup 1,” “backup 2,” and so on.)

After you do a backup, print a list of all the files that were backed up. To do this, select choice 5 on the menu. sysadmin will have you insert the backup diskettes one by one and will make a list of all the file names. The list will be in the file /tmp/backup.list. To print the list,

```
enter: pr /tmp/backup.list | lpr
```

Once the list is printed, you can erase this file to save space.

```
enter: rm /tmp/backup.list
```

To restore a file, select choice 6 on the menu. sysadmin will ask you to enter the full pathname of the file that you want to restore. Use the name as it was printed on your list. sysadmin will have you insert the backup diskettes one by one, until it finds the file you need.

### A PERSONAL BACKUP SYSTEM

#### Using the tar Program

The **tar** program backs up specific files or all the files in a specific directory. You can use **tar** to make a personal backup system. On a system with only a few users, the system manager may elect to maintain the overall backup system with **tar**, rather than with **sysadmin**.

**tar** stands for “tape archive.” It was designed for Unix systems that use tapes as auxiliary storage devices. The program has been updated, and it works fine with diskettes. However, you will notice that the terminology and messages associated with **tar** still reflect its origin as a tape backup program.

Before you use **tar**, you must have some formatted diskettes ready. [Formatting is explained in Chapter 15.] You do not need to make file systems on the diskettes.

**tar** is a complex program that has many options. In following sections, I will show you the options to use for the most common tasks. For an exhaustive description, see the Command Reference Manual.

The general form of the **tar** command is:

```
tar options special file...
```

where options are the options as explained below; **special** is the name of the special file that refers to the diskette drive; and **file** is the name of
the file you want to process. If you specify a directory, tar will back up all the files in that directory and in all its sub-directories.

Creating and Updating Backups

To create a new backup using tar, insert a formatted diskette into the drive and enter:

```
    tar -cfv special file...
```

The -c option creates a new backup. (c stands for "create.") The -f option means that the next argument is the name of the special file that refers to the diskette drive. (This is explained in more detail below.) The -v option displays informative messages as tar executes. (v stands for "verbose.")

For example, say that you want to create a backup on a high-capacity diskette in drive 0. You want to copy the files report and info.

```
    enter: tar -cfv /dev/fd0 report info
```

Remember, if you back up a directory, tar will copy all the files in the directory and in any sub-directories. For example, to back up all the files under the directory sales,

```
    enter: tar -cfv /dev/fd0 sales
```

If you want to back up all your files, specify your home directory. For example, the following command backs up all the files belonging to userid harley:

```
    tar -cfv /dev/fd0 /usr/harley
```

Here is a command that backs up all the files in your working directory:

```
    tar -cfv /dev/fd0 .
```

(Remember, a single period stands for your working directory.)

Whenever you create a new backup (that is, use the -c option), you will lose any files that are already on the diskette. If you want to add to a backup, use the -r or -u options:

```
    tar -rfv /dev/fd0 file...
    tar -ufv /dev/fd0 file...
```

The -r option adds files to the end of the backup. The -u option adds the files only if they are not there already, or if they have been modified since they were last backed up. (u stands for "update.")
For example, the following command adds the file `data` to the end of a backup:

```
tar -rfv /dev/fd0 data
```

This next command checks all the files that belong to userid `harley`. A file is added to the end of the backup, only if it is not already on the backup or if it has been modified:

```
tar -ufv /dev/fd0 /usr/harley
```

Creating a new backup (the `-c` option) corresponds to making periodic backups with the `sysadmin` program. Updating the backup (the `-u` option) corresponds to making a daily backup with `sysadmin`. If you use `tar` regularly, the best idea is to organize a regular pattern of backups. Use the pattern described in the `sysadmin` section. (Create a new backup every week; update the backup every day.)

**Listing and Restoring Files**

Once you have a backup, you can list the names of the files and you can restore files. To list the files,

```
tar -tnfv /dev/fd0
```

The `-t` option lists the names of the files on the backup. The `-n` option specifies that the backup is on a disk, not on a tape. This allows `tar` to search more quickly because it knows that it can skip over files that it does not want to read.

`tar` writes the list of file names to the standard output. Thus, you can use this form of `tar` at the beginning of a pipeline. For example, to print a list of files, use:

```
tar -tnfv /dev/fd0 | pr | lpr
```

To count the number of files, use:

```
tar -tnfv /dev/fd0 | wc -l
```

To restore one or more files, use:

```
tar -xnfv /dev/fd0 file...
```

The `-x` option copies files from the backup diskette to your working directory. (`x` stands for "extract.") Before you restore a file, you must change your working directory to what it was when you made the backup. This ensures that `tar` places the files in the correct directory.
For example, to restore the files *important* and *info*, change your working directory to what it was when you backed up the files and enter:

```
tar -xnfv /dev/fd0 important info
```

If you do not specify a file to restore, *tar* will restore all the files on the backup. For example:

```
tar -xnfv /dev/fd0
```

### Using Various Disk Drives

So far, all the examples of the *tar* command in this chapter have used the `-f` option with the name of a special file that refers to the diskette drive. For example, the following command creates a new backup, copying the file *report* onto the diskette in drive 0:

```
tar -cfv /dev/fd0 report
```

When you use the `-f` option, you can specify any one of a number of special files that refer to diskette drives. Most of the time you will use either `/dev/fd0` or `/dev/fd1`. These files refer to using drive 0 or drive 1, respectively, with a diskette that matches the type of drive (high-capacity or double-sided). For example, the following command is similar to the one above, except that it uses drive 1:

```
tar -cfv /dev/fd1 report
```

If necessary, you can use a double-sided diskette in a high-capacity drive. The FD section in Part 2 of the Command Reference Manual has a list of the special files you will need. Note: You cannot use a high-capacity diskette in a double-sided drive.

If you use *tar* a lot, you can set things up so that when you do not specify a special file, *tar* automatically uses a prearranged one.

Here is how you do it. Normally, *tar* will use the special file that comes after the `-f` option. However, if you leave out this option and the special file name, *tar* will automatically use the special file `/dev/mt1`. (*mt* stands for "mount," as in "mounting a tape.")

All you have to do is link this file to the special file that you use most of the time. For example, if you usually use `/dev/fd0`, you can make the link by entering:

```
l in /dev/fd0 /dev/mt1
```

This should be done by the system manager. (You must be superuser to
create this particular link.)

Once this is set up, your tar commands will be shorter and easier to enter. For example, say that you make the link shown above. Instead of the commands:

```bash
tar -cfv /dev/fdo report
tar -ufv /dev/fdo report
tar -tnfv /dev/fdo

tar -xnfv /dev/fdo report
```

you can use:

```bash
tar -cv report
tar -uv report
tar -tnv

tar -xnv report
```

In each case, you can omit the -f option and the name of the special file.

### USING DOS FILES

#### The dos Commands

Xenix is not the only operating system that you can use to run the IBM PC AT computer. The Disk Operating System (DOS) can run any of the IBM Personal Computers. In fact, if you are using a PC AT, you can install both Xenix and DOS and switch from one to the other.

Xenix has a set of commands, called the dos commands, that allow you to access DOS files from a diskette. This provides a way to pass information between Xenix and a DOS system. The following sections describe these commands. If you are not interested in DOS, you can skip the rest of this chapter.

With the dos commands, you specify a file just as you do with DOS: a device name, followed by a colon, followed by a file name. You cannot use the DOS wildcard characters.

You can use the following device names:

- **X**: a high-capacity diskette in drive 0
- **Y**: a high-capacity diskette in drive 1
- **A**: a double-sided diskette in drive 0
- **B**: a double-sided diskette in drive 1

Or you can use the name of a special file that refers to a diskette.
For example, say that you want to specify the DOS file `program.bas`, which is on a high-capacity diskette in drive 0. You can use either of:

\[
\begin{align*}
&X:program.bas \\
&/dev/id0:program.bas
\end{align*}
\]

If you specify a path before the name of a file, you must separate the directory names by a slash (/), as you do with Xenix; rather than by a backslash (\), as you do with DOS.

For example, say that you are using a double-sided DOS diskette in drive 1. The diskette has a directory `accounts`, which has a sub-directory `paid`, which has a file `january`. To refer to the file in a `dos` command you would use:

\[
B:accounts/paid/january
\]

The `dos` commands act only on files that are on a diskette that has been formatted by DOS. They will not work with a Xenix formatted diskette; nor will they work with files on the fixed disk.

**Working with Files on a DOS Diskette**

To copy files to and from a DOS diskette, use the `doscp` and `doscat` commands. To erase files from a DOS diskette, use the `dosrm` command.

The `doscp` command has two forms. The first form copies one text file to another. The syntax is:

```
   doscp file1 file2
```

where `file1` is the name of the text file that is to be copied and `file2` is the name of the target file. One of these must be a Xenix file and the other must be a DOS file on a DOS diskette.

The following example copies the Xenix file `report` from the working directory to the file `report.txt` on a high-capacity DOS diskette in drive 0:

```
   doscp report X:report.txt
```

This next example copies the DOS file `info.txt` from the `accounts` directory on the same DOS diskette to the Xenix file with the name `/usr/acme/accounts/info`:

```
   doscp X:accounts/info /usr/acme/accounts/info
```

The second form of the `doscp` command copies one or more text files to a directory. The syntax is:
**doscp file... directory**

where file is the name of a text file to be copied, and directory is the name of the directory into which the copy is to be put.

You can copy Xenix files to a DOS directory or DOS files to a Xenix directory. For example, to copy the Xenix files *info* and *report* from the working directory to the directory *accounts* on device *X:*,

```
enter: doscp info report X:accounts
```

If you want to copy the same files to the root directory of *X:*,

```
enter: doscp info report X:
```

To copy the files *jan.txt*, *feb.txt*, and *mar.txt* from the same DOS diskette to the Xenix working directory,

```
```

(Remember, a single period stands for the working directory.)

The **doscat** command acts much like the **cat** command. **doscat** will combine and copy one or more DOS files from a DOS diskette to the standard output. The syntax is:

```
doscat dosfile...
```

where *dosfile* is the name of a DOS file on a DOS diskette.

For example, to display the DOS file *accounts/info* on device *X:*,

```
enter: doscat X:accounts/info | more
```

To combine the DOS files *jan.txt*, *feb.txt*, and *mar.txt* from the same diskette and store that result as the Xenix file *months*,

```
enter: doscat X:jan.txt X:feb.txt X:mar.txt > months
```

To remove files from a DOS diskette, use the **dosrm** command. The syntax is:

```
dosrm dosfile...
```

where *dosfile* is the name of a file on a DOS diskette. For example, to erase the files *report* and *info* from a high-capacity DOS diskette in drive 0,

```
enter: dosrm X:report X:info
```

**Working with Directories on a DOS Diskette**

There are several commands that you can use to work with directories on a DOS diskette. To display the files in a directory, use
dosdir and dosls. The syntax is:

\begin{enumerate}
\item dosdir dosdirectory...
\item dosls dosdirectory...
\end{enumerate}

where dosdirectory is the name of a directory on a DOS diskette.

The dosdir command displays the file names in the standard DOS format. The dosls command displays the files in the same format as the Xenix ls command.

Here are two examples, each of which displays the names of the files in the directory /accounts/monthly on a high-capacity DOS diskette in drive 0:

\begin{enumerate}
\item dosdir X:/accounts/monthly
\item dosls X:/accounts/monthly
\end{enumerate}

To create a directory on a DOS diskette use the dosmkdlr command. The syntax is:

\begin{enumerate}
\item dosmkdlr dosdirectory...
\end{enumerate}

where dosdirectory is the name of the new directory.

The following command creates the directory /accounts on the high-capacity DOS diskette in drive 0:

\begin{enumerate}
\item dosmkdlr X:/accounts
\end{enumerate}

To remove a directory from a DOS diskette, use the dosrmdlr command. The syntax is:

\begin{enumerate}
\item dosrmdlr dosdirectory...
\end{enumerate}

where dosdirectory is the name of the directory on a DOS diskette. For example, the following command removes the directory /accounts from the high-capacity DOS diskette in drive 0:

\begin{enumerate}
\item dosrmdlr X:/accounts
\end{enumerate}
CHAPTER 20
PROGRAMMING THE SHELL—BEGINNING CONCEPTS

INTRODUCTION
THE BASIC CONCEPTS
USING VARIABLES
A FEW IMPORTANT FEATURES
INTRODUCTION

In Chapter 1, I told you that Xenix is a powerful operating system. Throughout this book you have learned commands to perform many different tasks. However, the power of Xenix comes from more than its large selection of commands; the power comes from the fact that you can combine commands into programs that you can use at your convenience. In other words, you can design your own custom commands.

In this chapter, you will learn the basics of designing your own commands. In Chapter 21, you will learn some advanced features. The skills you learn from these two chapters will open the doors to the power of Xenix.

New Words in This Chapter

SHELL SCRIPT
a file of commands that is to be executed by the shell

SHELL PROCEDURE
same as SHELL SCRIPT

SHELL FILE
same as SHELL SCRIPT

COMMENT
within a shell script, a descriptive line that is ignored by the shell when the script is executed

EMPTY
describes a variable that has not been assigned a value

POSITIONAL PARAMETER
within a shell script, a variable that is assigned the value of one of the arguments that was specified when the script was executed

USER-DEFINED VARIABLE
a variable that is created within a shell script

EXPORT
within a shell script, to pass the value of a variable to scripts that are subsequently executed
GLOBAL VARIABLE
a variable that has been exported; hence, its value is available to more than one shell script

LOCAL VARIABLE
a variable that has not been exported; hence, its value is available only within one shell script

THE BASIC CONCEPTS

Programming the Shell

As you know, the shell is the program that reads and interprets your commands. Generally speaking, you enter commands from the keyboard, one by one. However, you can create a file of commands and have the shell read and execute them, just as if you had typed them.

In effect, what you are doing is programming the shell. The idea is to design a sequence of commands that, when executed, will have a desired effect.

A file of commands that is to be executed by the shell is called a SHELL SCRIPT. (Scripts, in general, are discussed in Chapter 14.) Shell scripts are sometimes called SHELL PROCEDURES or SHELL FILES. In this chapter, when I say "script," I mean "shell script."

A shell script may contain any Xenix commands that you can enter from the keyboard. For example, here is a simple shell script. Its purpose is to display the time and date, followed by the names of all the files in the working directory.

date
lc

Shell scripts can be simple, as is the one above; however, you can also design scripts to automate sophisticated tasks. Toward this end, Xenix has a number of commands that were designed especially for shell scripts. With these commands, the Xenix system becomes a powerful programming language.

To understand the shell and its nuances completely, you would need a background in programming and a good deal of experience. However, you do not have to be an expert programmer to write useful shell scripts. In this chapter, I will introduce you to the most important concepts and commands. For more detailed information see Chapter 5 of the Basic Operations Guide, and the sh entry in Part 1 of the Command Reference Manual.
This chapter describes how to program the Bourne shell. If you have the Software Development System, you may want to use the C-Shell. (The different shells are described in Chapter 3.) For detailed information on the C-Shell, see Chapter 10 of the Software Development Guide and the csh entry in Part 1 of the Software Command Reference Manual.

Using a Shell Script

Here are the steps that you follow to use a shell script:

1. Create the script by using the vi editor.
2. Make the script executable by using the chmod command.
3. Execute the script by entering its name.

Here is an example: Say that you want to make a script to display the time and date, followed by the names of the files in the working directory. The script is to be called filecheck. First, use vi and create a text file named filecheck that contains:

```
date
lc
```

As you learned in Chapter 17, when you create a new file, Xenix automatically gives it the following permissions:

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

In other words: the owner can read and write to the file; everyone else can only read it. Before you can use a shell script, you must make it executable. The following command will do this for the file in our example:

```
chmod 755 filecheck
```

filecheck will now have the permissions:

```
-rw-xr-xr-x
```

In other words, the owner can read, write, and execute the file; everyone else can read and execute it. (The chmod command is described in Chapter 17.)

Once a file is executable, you can use it by entering its name. For example, to execute the commands in filecheck,

```
enter: filecheck
```
Xenix will execute the commands in the script, just as if you had typed them at the keyboard.

Making a Shell Script Understandable

It is important that you write your shell scripts so that they can be easily understood. There are two techniques that you can use. First, you can include COMMENTS. A comment is a descriptive line that is ignored by the shell.

In a shell script, any characters that come after a # character are ignored. Thus, you can put in a comment by preceding it with a #. Here are two examples:

```
# This line is ignored by the shell.
date # This command displays the time and date.
```

The first line consists of only a comment. The second line consists of the `date` command followed by a comment.

When you write a shell script, it is customary to include a comment at the beginning giving the name and purpose of the script. As well, it is a good idea to preface each group of commands with a comment that explains what the commands do. Here is a simple example:

```
# filecheck: shell script to check files in working directory
# display the date
date
# display the files in the working directory
lc
# finished
```

The second technique that you can use to write understandable shell scripts depends on the fact that the shell ignores `space` and `tab` characters at the beginning of a line. As well, the shell ignores empty lines. Thus, you can indent and space the commands to reflect the logic of the program. For example:

```
# filecheck: shell script to check files in working directory
# display the date
date
# display the files in the working directory
lc
# finished
```

Unless you are an experienced programmer, it may seem silly to
include so many comments and empty lines. You must understand that the above example is contrived; it is a simple, two-command shell script. Most scripts are more complicated. In such cases, the comments and spacing make an enormous difference. You will notice this as you read the examples in this chapter.

You will occasionally want to use commands that are so long that they do not fit on one line. The shell allows you to break such commands into more than one line. The rule is that if a command ends with a backslash (\), the shell considers it to be continued on the next line.

Here is an example of an `lpr` command that prints many files. Notice how indentation makes the command easy to understand.

```
lpr report.jan report.feb report.mar report.apr
   report.may report.jun report.jul report.aug
   report.sep report.oct report.nov report.dec
```

Using Shell Scripts to Perform Common Tasks

The first important use of shell scripts is to perform common tasks that involve entering the same commands repeatedly. For example, you may have some commands that you perform every time you start work on a particular project. In this case, you can put the commands in a script.

Here is an example of such a script called `begin`:

```
# begin: shell script to perform beginning tasks
# check the mail
mail

# change working directory to 'project'
cd project
echo
echo -n 'Working directory is: 'pwd

# display the files in the working directory
echo
echo 'Files are:'
lc

# display the time and date
echo
echo -n 'Date is: 'date

# finished
```
Here is some sample output from this script:

No messages.

Working directory is: /usr/harley/book/project

Files are:
addresses data report.1 report.2

Date is: Thu Aug 22 18:45:29 PDT 1985

This shell script illustrates how you can use the `echo` command to write messages to the terminal. Notice that when you use `echo` with no arguments it displays an empty line.

Also, notice the effect of the `-n` option. Normally, `echo` ends each line with a newline character. When you use the `-n` option, `echo` leaves out the newline. This means that further output to the terminal continues on the same line. Thus, where the commands:

```
  echo 'Working directory is: '
  pwd
```

would generate two lines of output, the commands:

```
  echo -n 'Working directory is: '
  pwd
```

display their output on the same line.

Using Shell Scripts to Perform Awkward Tasks

Occasionally, you will run up against a command that is awkward to enter. If you only use it once, this is not a problem. But if you use the command more than once, you are better off putting it into a shell script.

Here is an example. In Chapter 15, you learned how to use the `mount` command to mount a file system contained on a diskette. If you use a diskette frequently, you will soon find that the `mount` command is a particularly awkward one to enter.

Here is a script that you might use to do the mounting for you. It mounts the diskette in drive 0, using the directory `/usr/harley/backup`.

```
# mbackup: shell script to mount backup diskette
# mount the diskette in drive 0
   /etc/mount /dev/fd0 /usr/harley/backup
# finished
```
Here is a script to do the dismounting:

```
# ubackup: shell script to dismount backup diskette
# dismount the diskette in drive 0
  /etc/unmount /dev/fd0

# finished
```

Now, whenever you want to mount the diskette, you only need enter:

```
mbackup
```

When you want to dismount the diskette, use:

```
ubackup
```

### USING VARIABLES

#### Introduction to Variables

So far, you have only seen shell scripts which contain commands that you might enter at the terminal. You will now learn about some of the features that are used only within scripts.

The most important of these features is the use of variables. In Chapter 6, you learned how to use variables with the `bc` (calculator) program. In that chapter, you learned that a variable is a quantity with a name that can take on different values.

When you use `bc`, variables take on numeric values. Within a shell script, the only value that a variable can have is that of a character string. Another way to think of it is that a variable stands for a particular character string. For example, the variable `filename` might have the value `/usr/harley/book/outline`.

#### Positional Parameters

There are two types of variables that you can use in a shell script. The first type are called POSITIONAL PARAMETERS. Positional parameters allow you to pass information to the shell script by using arguments. There are ten positional parameters. Their names are `$0`, `$1`, `$2`, `$3`, `$4`, `$5`, `$6`, `$7`, `$8`, and `$9`.
When you enter an ordinary command, you type the name of the command, followed by any arguments. For example, the command:

```
lc report outline data
```

consists of the name `lc`, followed by three arguments. The command uses the information in the arguments to carry out its task.

When you design a shell script, you will often want to be able to use information that is passed in the form of arguments. To allow you to do this, the shell assigns the value of each argument to a particular positional parameter. The first argument is called `$1`, the second argument is called `$2`, and so on. The name of the script itself is called `$0`.

From within the script, you can refer to these values by using their names. Here is an example of a script that illustrates this: The script combines several files and then counts the total number of lines.

```
# fllestat: shell script to combine files and count lines
# combine files, then count lines
  cat $1 $2 $3 $4 $5 $6 $7 $8 $9 | wc -l
# finished
```

When you use this script, you specify as arguments the names of the files that you want to process. The script refers to the arguments as `$1` through `$9`. This allows you to specify up to nine file names. Any positional parameters that are not specified are considered to be empty. For example, say that you execute the script by entering:

```
fllestat report data outline
```

The positional parameters are assigned the values:

```
$0   fllestat
$1   report
$2   data
$3   outline
$4 through $9   empty
```

The shell substitutes for the positional parameters before it executes the command. The result is that the following command is executed:

```
cat report data outline | wc -l
```

Since `$4` through `$9` are empty, they vanish from the command line.

Within a script, the variable `$#` is automatically set to be the
number of the highest nonempty positional parameter. Thus, if a user specifies five arguments, $# will have the value 5. You can test $# at the beginning of a script to make sure that the user has specified the right number of arguments.

An Example Using Positional Parameters

The importance of positional parameters is that they allow you to write shell scripts to perform general tasks.

For example, say that you want to search a directory and its subdirectories for a file. In Chapter 9, you learned how to use the `find` command to do this. However, this command has an awkward syntax. Not only that, you usually want to search only your own directories, and it is inconvenient to have to specify the name of your home directory each time.

Here is a shell script that searches all the directories of one user, looking for a file. In this case, the userid is `dianne`. The name of the file is passed as an argument and is referred to as $1.

```
# findfile: shell script to search user's directories
# search all of the user's directories for a file
# whose name is passed as $1
    find /usr/dianne -name $1 -print

# finished
```

The power of this script lies in the fact that you can use it to search for any file. For example, to search for the file `report`, use:

```
findfile report
```

Shifting Positional Parameters

You can manipulate positional parameters by using the `shift` command. The syntax is:

```
shift
```

At any time, you can access up to nine positional parameters by using $1 through $9. The `shift` command shifts these values one position to the left. In other words, $1 takes on the value of $2; $2 takes on the value of $3; and so on. The original value of $1 is lost.
Here is an example of a script that demonstrates how this command works. The script is called `shift.text`.

```bash
#!/bin/bash

# shift.text: shell script to demonstrate the shift command
# shift the positional parameters four times; display them
# before and after each shift operation
for _ in {0..3}; do
    echo $0 $1 $2 $3 $4 $5 $6 $7 $8 $9; shift
done

# finished
```

Say that you execute this script as follows:

```bash
shift.test 1 2 3 4 5 6 7 8 9 10 11
```

The output is:

```
shift.test 1 2 3 4 5 6 7 8 9 10 11
shift.test 2 3 4 5 6 7 8 9 10
shift.test 3 4 5 6 7 8 9 10 11
shift.test 4 5 6 7 8 9 10 11
shift.test 5 6 7 8 9 10 11
```

Notice three things: First, although the `shift` command changes the value of `$1` through `$9`, it does not change `$0` (the name of the script).

Second, a script may be passed more than nine arguments. The `shift` commands shifts all the arguments, even those to the right of `$9`. Whenever there are more than nine arguments left, the `shift` command shifts the tenth argument into `$9`. This allows you to use many arguments.

Third, if there are nine or fewer arguments, an empty string is shifted into `$9`.

By itself, the `shift` command is not that useful. It is generally used with other commands in order to process a series of arguments. (You will see examples of this in Chapter 21.)

**User-Defined Variables**

The second type of variable you can employ is called a **USER-DEFINED VARIABLE**. It differs from a positional parameter in that it
does not exist until you create it (hence, the name "user-defined").

When you create a user-defined variable, you must give it a name. The name can be any sequence of letters, digits, or underscore characters (-) that does not start with a digit. To create a user-defined variable, all you have to do is assign it a value. The syntax for doing so is:

\[ \text{varname} = \text{string} \]

where \text{varname} is the name of the variable and \text{string} is a character string that is to be the value of the variable.

For example, to create a user-defined variable called \text{file} with the value /usr/harley/report, use:

\[ \text{file} = /usr/harley/report \]

Notice that there are no \text{space} characters on either side of the equals sign.

The meaning of the equals sign is as follows: Evaluate the expression on the right and assign its value to the variable name on the left. You can use an assignment statement any time you want to change the value of a user-defined variable. (You cannot use a positional parameter to the left of an equals sign.)

When you name a user-defined variable, the shell distinguishes between upper-case and lower-case letters. For example, the names \text{file} and \text{FILE} represent two distinct variables.

To use the value of a user-defined variable, precede the variable name by a dollar sign ($). Here is an example of a few lines from a shell script:

\begin{verbatim}
   today=Monday
   echo $today
   echo today
   echo '$today'
\end{verbatim}

The output would be:

\begin{verbatim}
Monday
today
$today
\end{verbatim}

In the first echo command, the value of the user-defined variable \text{today} is substituted. In the second echo command, no substitution takes place because there is no dollar sign. In the last command, the dollar sign is taken literally because it is within single quotes; thus, no substitution takes place.
Working with Variables

There are a few rules that you need to know in order to work with variables. These rules apply to both positional parameters and user-defined variables.

Any variable that has not been assigned a value is said to be EMPTY. Xenix considers all empty variables to have the value of a string with no characters.

For example, consider the commands:

```
today=Monday
tomorrow=Tuesday
echo $today $yesterday $tomorrow
```

The dollar sign at the beginning of `$yesterday` means to substitute the value of the variable `yesterday`. But `yesterday` is empty, so no characters are substituted. The result of the `echo` command is:

```
Monday Tuesday
```

You can use the value of a variable as part of a longer character string. For example, the commands:

```
suffix=ing
echo writ$suffix programm$suffix
```

produce the output:

```
writing programming
```

If the variable name is directly followed by a string, you can enclose the name in braces to separate it. For example:

```
day1=Mon
day2=Tues
echo Today is ${day1}day
echo Tomorrow is $day2day
```

produces:

```
Today is Monday
Tomorrow is
```

In the first `echo` command, the value of the variable `day1` is substituted as part of a larger string. In the second `echo` command, the shell considers `$day2day` to be a reference to the variable `day2day`. Since this is empty, no characters are substituted.
An Example Using User-Defined Variables

Here is an example of a shell script that demonstrates the use of user-defined variables. The script is called sales.report. It might be used by a sales manager at the end of each week to generate a report on one of his or her salespersons. Here is how it works:

Each salesperson has a Xenix userid and is required to keep a file sales in a directory called info. Each line in sales has the name of one customer, followed by the number of products sold to that customer in the current week.

The shell script uses this file to generate a report. The userid of the salesperson is passed as the first positional parameter ($1).

```bash
# sales.report: shell script to display a sales report
# variables:
#    userid -- the userid of a salesperson
#    customers -- the customer information file
# set the names of the files to be used
userid=$1
customers=/usr/$userid/Info/sales

# display the heading for the report
echo -n 'Report on salesperson: '$userid
echo
echo -n ' for the week ending: '
date
echo

# display the number of customers, followed by the
# the customer file in alphabetical order
echo -n 'Number of customers: '
cat $customers | wc -l
echo
echo 'Customer List:'
echo
sort $customers

# finished
```

Say that a salesperson with a userid of evelyn has the following information in her file /usr/evelyn/Info/sales:
To generate a report on these sales,

enter: `sales.report evelyn`

The report would look like this:

```
Report on salesperson: evelyn
            for the week ending: Fri Aug 23 17:23:10 PDT 1985

Number of customers: 5

Customer List:
California Widget Traders -- 350 widgets sold
Universal Widget Company -- 200 widgets sold
International Widget -- no sales
PFA Enterprises -- 950 widgets back ordered
Pushbutton Inc. -- no sales
Universal Widget Company -- 200 widgets sold
```

**Using a Script from within a Script**

You can use a shell script from within another shell script. When you do, the variables in the first script are put into temporary storage. They are not passed to the second script unless you specifically indicate it. (This is explained in the next section.) Thus, when you write a script that calls another script, you do not have to worry about accidentally using the same variable names.

Here is an example. The last section contains a shell script called `sales.report` that displays a report on a salesperson. The userid of the salesperson is passed to the script as its first argument. Say that you wanted to use this script at the end of every week to print reports on six salespersons with the userids `evelyn`, `george`, `sam`, `gertrude`, `siena`, and `arnie`.

You can do this by entering six commands of the form:

```
sales.report evelyn | pr | lpr
```

However, since you do this every week, you are better off making a shell script to do everything for you.
Using Variables  313

# print.report: shell script to print weekly sales report

# print one report for each salesperson
  sales.report evelyn  |  pr  |  lpr
  sales.report george  |  pr  |  lpr
  sales.report sam     |  pr  |  lpr
  sales.report gertrude|  pr  |  lpr
  sales.report sheila  |  pr  |  lpr
  sales.report arnie   |  pr  |  lpr

# finished

Now, at the end of each week, all you have to do is enter:

  print.report

Since the variables are kept separate, it does not matter that you use the same script more than once. Nor would it matter if the script sales.report used another script with the same variables.

Passing Variables to Another Script

When you use a script from within another script, none of the variables are passed between the scripts. This ensures that one script cannot meddle with another. However, there are times when you do want to share variables.

You can do so by telling the shell that certain variables are to be EXPORTED, or passed, to any scripts that are started from the original script. To do this, use the command:

  export varname

where varname is the name of a user-defined variable.

For example, say that you design a complex system of scripts to perform backups. To start the system, you execute a script called backup. This script starts other scripts, which start other scripts, and so on. Throughout it all, every script uses the name of the backup device, which does not change.

Now, you can pass the device name as a positional parameter from one script to the next. However, it is simpler to assign the name to a variable from within backup and then export it to all the other scripts. The commands might look like this:

  DEVICE=/dev/fd0
  export DEVICE
A variable that is passed between scripts in this manner is called a GLOBAL VARIABLE. A variable that is confined to one script is called a LOCAL VARIABLE. It is customary to use upper-case letters for the names of the global variables. This allows them to be easily identified.

It is important to realize that the shell exports a copy of the variable, not the variable itself. If a script changes a variable that has been exported to it, the change is not passed back to the original scripts.

For example, within the script backup, the global variable DEVICE will not be permanently changed by another script.

Generally speaking, global variables should be used sparingly. Because they can be set and reset by various scripts, they can lead to a great deal of confusion. You should only use them when it is impossible or inconvenient to use positional parameters.

A FEW IMPORTANT FEATURES

Command Substitution

Xenix allows you to use the standard output of a command as a character string. To do so, enclose the command in back-quotes. The shell will execute the command and replace it by its standard output. Here is an example:

```
    echo The time is 'date'
```

Before the shell executes this command, it replaces the string ‘date’ with the output of the date command. Thus, the result of the command looks like this:

**The time is Fri Aug 23 20:16:14 PDT 1985**

This is especially useful in shell scripts because it allows you to manipulate the output of a command. In particular, you can assign it to a variable and recall it as necessary.

Here is an example: The following script displays general information about the status quo:

```
# status: shell script to display general information
# display the time, logname, name of the working directory, # and number of files in the working directory
    echo The time is 'date'.
    echo Your userid is 'logname'.
    echo Your working directory is 'pwd'.
    echo It has 'lc -1 | wc -1' files.

# finished
```
The output looks like this:

```
Your userid is harley.
Your working directory is /usr/harley/book.
It has 24 files.
```

Notice the last `echo` command. It determines the number of files by listing them, one to a line, and then counting the number of lines.

When you substitute for a command, it generates a string that you can assign to a variable. Here is a shell script that does this. The script is called `print.info`, and it prints a copy of every file in the directory `/usr/acme/sales/info`. To do this, it must change the working directory. The name of the original working directory is saved in the variable `dir` for later restoration.

```
# print.info: shell script to print all the files in /usr/acme/sales/info

# variables:
#   dir -- the name of the original working directory
#   save name of working directory
  dir=`pwd`

# print the files
  cd /usr/acme/sales/info
  pr * | lpr
  echo 'lc -1 | wc -1' files have been printed.

# restore the working directory
  cd $dir

# finished
```

Notice the `echo` command. When the printing is finished, this command uses the output of the `wc` command to display an informative message.

Quoting

So far, you have learned two ways to quote strings. First, you can enclose a string in single-quotes. Second, if the string consists of only one character, you can precede it by a backslash.

Consider the following examples:

```
echo time; date
echo time ';' date
echo time \; date
```
The first example consists of two commands separated by a semicolon. The second and third examples are each one command. This is because the semicolon is quoted; thus, it is taken literally and loses its special meaning to the shell. The output from both the second and third commands is:

```
$ time ; date
```

Except for the fact that a backslash can only quote one character, it has the same effect as single-quotes. However, there is one more way to quote a string: using double-quotes. And this has a slightly different effect.

When you quote with single-quotes or a backslash, all characters are taken literally. However, with double-quotes, the dollar sign and back-quote characters retain their special meaning. Thus, you can use variables and command substitution.

Here are two examples that you might use within a shell script: Say that the value of the variable `file` is `info`. Consider the commands:

```
$ echo 'The time is `date`; The file is $file.'
$ echo "The time is `date`; The file is $file."
```

In the first example, all the characters within the single-quotes are taken literally. The output is:

```
The time is `date`; The file is $file.
```

However, in the second example, the back-quotes and the dollar sign retain their special meaning (although the semicolon does not). The output looks like:

```
The time is Fri Aug 23 22:50:02 PDT 1985; The file is info.
```

Note: Do not confuse the back-quote character with the single quote and double-quote. Back-quotes are used to indicate command substitution; they have nothing to do with quoting.

### Grouping Commands Using Braces

There are two ways that you can group a sequence of commands so that they are treated as one long command. The first way is to enclose the commands in braces. When you do, the shell performs the commands in sequence but combines the output. Here is an example:

```
{ echo "User report for `date`."; who; } | pr | lpr
```
The output of the `echo` and `who` commands is combined and piped to the `pr` and `lpr` programs. The result is a printed report showing which userids are logged in. Notice the use of command substitution with the `date` command to insert the time and date.

When you use braces, you must follow two rules: First, you must place a `space` after the left brace; second, you must place a semicolon before the right brace.

Although you can use braces when you enter commands from the keyboard, they are especially useful in a shell script. Consider the following script, named `user.report`, that prints a more elaborate report of who is logged in to the system. Notice that the braces can enclose commands that are on separate lines.

```bash
# user.report: shell script to print report of current users
# print the date, number of users, and sorted list of userids
{   echo "Report of users for 'date'."
    echo "There are 'who I wc -1' users logged in."
    who | sort;} | pr | lpr
# finished
```

This script produces output that looks like:

Aug 24 14:36 1985 Page 1
There are 3 users logged in.
harley tty01 Aug 24 10:36
ilsa console Aug 24 14:01
michael tty00 Aug 24 15:23

Grouping Commands Using Parentheses

The second way that you can group commands is by enclosing them in parentheses. The parentheses indicate that the commands are to have no permanent effect on variable values or the name of the working directory.

Here is an example: Your working directory is `/usr/sam/work`. You want to display four files in the directory `usr/mary/reports/info`. One way to do this is to enter the commands:
318 Programming the Shell—Beginning Concepts

```bash
more /usr/mary/reports/info/file1
more /usr/mary/reports/info/file2
more /usr/mary/reports/info/file3
more /usr/mary/reports/info/file4
```

An easier way is to change the working directory, display the files, and restore the working directory:

```bash
cd /usr/mary/reports/info
more file1 file2 file3 file4
cd /usr/sam/work
```

However, you can get the same effect by enclosing the first two commands in parentheses; this serves to localize the effect of the `cd` command.

```bash
(cd /usr/mary/reports/info; more file1 file2 file3 file4)
```

Unlike when you use braces, you do not need a `space` after the left parenthesis; nor do you need a semicolon before the right parenthesis.

Parentheses will also protect the values of variables. For example, the following commands:

```bash
word=goodbye; echo $word
(word=hello; echo $word)
```

display:

```bash
goodbye
hello
goodbye
```

The parentheses have preserved the value of the variable `word`.

Here is an example of a shell script that is an improvement on the script called `print.info` from a few sections back. The improvement comes from using parentheses to preserve the working directory. This ensures that when you execute the script, it does not have the side effect of changing your working directory.

```bash
# print.info: shell script to print all
#             the files in /usr/acme/sales/info

# print the files
   (cd /usr/acme/sales/info
    pr * | lpr
    echo 'lc -1 | wc -l' files have been printed.)

# finished
```
CHAPTER 21

PROGRAMMING THE SHELL—ADVANCED CONCEPTS

INTRODUCTION
PERFORMING TESTS WITHIN A SHELL SCRIPT
CONTROL FLOW
SETTING UP YOUR WORKING ENVIRONMENT
INTRODUCTION

In the last chapter, you learned how to create and execute shell scripts. In this chapter, you will learn how to use commands that allow you to take full advantage of the shell as a programming environment. You will be able to design scripts to choose between alternatives and to execute commands repeatedly. The chapter ends with a description of how you can customize your working environment.

New Words in This Chapter

CONTROL FLOW
the logic that describes the order in which the instructions in a program are executed

SEARCH PATH
the list of directories that the shell searches when it is looking for a program

PERFORMING TESTS WITHIN A SHELL SCRIPT

How It Works

The shell provides you with a way to test for various conditions from within a script. A test is not used by itself; rather it is part of another command. In the following sections, you will learn how to construct a test. You will then learn how to use the commands that require a test.

A test is a construction that, when evaluated by the shell, will yield a result that can be considered to be either true or false. For example, you might test to see if the variable file has the value /usr/acme/report.

A test consists of a condition enclosed by square brackets. Each square bracket must have a space on either side. Here is an example:

[ $file = /usr/acme/report ]

There are many different conditions that you can test for. I will describe only the most important ones. For an exhaustive list, see the test entry in Part 1 of the Command Reference Manual.
Performing Tests within a Shell Script

Testing Character Strings

The most useful tests determine if two character strings are the same or different. To test if two strings are the same, use:

```
[ string1 = string2 ]
```

where `string1` and `string2` stand for two strings. For example, to test if the variable `words` has the value `hello to you`, use:

```
[ $words = 'hello to you' ]
```

(Remember, variables contain character strings.) To test if the variable `words` has the same value as the variable `sentence`, use:

```
[ $words = $sentence ]
```

Sometimes you want to determine if two strings have different values. In this case, use:

```
[ string1 != string2 ]
```

(Think of `!=` as meaning "not.") For example, to test if the variable `words` has a different value than the variable `sentence`, use:

```
[ $words != $sentence ]
```

If you want to see if a string is not empty, use the test:

```
[ string ]
```

For example, to test if positional parameter `$1` is not empty, use:

```
[ $1 ]
```

Testing File Names

There are several useful tests you can use that involve files. You can test if a string contains the name of an existing ordinary file by using:

```
[ -f string ]
```

For example, to test if the variable `source` contains the name of an existing ordinary file, use:

```
[ -f $source ]
```

To test for the opposite condition, use:

```
[ ! -f string ]
```
For example, to test if the file `/usr/acme/info` is not an existing ordinary file, use:

```
[ ! -f /usr/acme/info ]
```

To test if a string contains the name of an existing directory, use `-d` instead of `-f`. For example, to test if the value of the positional parameter `$1` contains the name of an existing directory, use:

```
[ -d $1 ]
```

To test if the variable `dir` does not contain the name of an existing directory, use:

```
[ ! -d $dir ]
```

As I mentioned above, each condition that you test is evaluated by the shell as being either true or false. The value of the condition affects the action of a control flow command. Occasionally, you will want to specify a test that is always true or a test that is always false. In this case, you can use the commands `true` and `false`.

### Testing Numeric Values

If you are working with strings that contain numbers, you can perform numeric tests. Use the following tests to compare the numeric values of two strings (designated as `string1` and `string2`). Of course, these tests do not make sense if the strings do not contain numeric characters.

```
[ string1 -eq string2 ]     equal
[ string1 -ne string2 ]     not equal
[ string1 -lt string2 ]     less than
[ string1 -le string2 ]     less than or equal
[ string1 -gt string2 ]     greater than
[ string1 -ge string2 ]     greater than or equal
```

For example, say that the variables `boys` and `girls` contain numeric characters. To test if `boys` is greater than or equal to `girls`, use:

```
[ boys -ge girls ]
```

### CONTROL FLOW

The CONTROL FLOW of a program is the logic that describes the order in which the instructions of the program are executed. The shell
provides commands that let you structure the control flow of your scripts in a variety of ways. You can write scripts that test various conditions, that choose between alternatives, or that execute commands repeatedly.

The if Command

The if command allows you to test a condition and, based on the result of the test, choose from one or more alternatives. You can use this command to make decisions in a shell script.

You can use the if command in several ways. The most basic way uses the following syntax:

```
if test
 then commands
 fi
```

where test is a test of a condition, and commands is a sequence of one or more commands. The shell will perform the test. If not, the shell will ignore the commands. The fi is necessary to mark the end of the commands following the then.

Here is an example of a shell script that uses this form of the if command: This script accepts one argument, which should be the name of a file, and tells you if the file is an ordinary file or a directory.

```
# filetype: shell script to determine the type of a file
# arguments:
#   $1 -- the name of a file
# if the file is an ordinary file, display a message
 if [ -f $1 ]
  then echo "$1 is an ordinary file."
 fi

# if the file is a directory, display a message
 if [ -d $1 ]
  then echo "$1 is a directory."
 fi

# finished
```

The second form of the if command allows you to specify one or more commands to be executed if the test is not true. The syntax is:
If test	hen commands
else commands
fi

In this case, the commands after the else are executed if the test is false.
The following shell script illustrates this form of the if command. The script is an improvement on the last one. The improvement is that the script lets you know if the argument is not the name of an ordinary file or of a directory.

# filetype: shell script to determine the type of a file
# arguments:
#   $1 -- the name of a file
# display a message saying whether or not the file is
# an ordinary file
  if [ -f $1 ]
    then echo "$1 is an ordinary file."
    else echo "$1 is not an ordinary file."
  fi
# display a message saying whether or not the file is
# a directory
  if [ -d $1 ]
    then echo "$1 is a directory"
    else echo "$1 is not a directory."
  fi

# finished

You can use more than one command after the then or the else. You can even use another if command as this next script illustrates:

# filetype: shell script to determine the type of a file
# arguments:
#   $1 -- the name of a file
# display a message saying if the file is an ordinary file, a directory or neither
  if [ -f $1 ]
    then echo "$1 is an ordinary file."
    else if [ -d $1 ]
      then echo "$1 is a directory"
      else echo "$1 is not an ordinary file or directory."
      fi
  fi

# finished
Notice that the commands that belong to the second `if` are indented a few extra positions. This makes the control flow easy to understand when you read the script.

The last form of the `if` command allows you to perform another test if the first test is not true. The syntax is:

```bash
if test
  then commands
else test
  then commands
else commands
fi
```

Think of `else` as meaning "else if." Thus, the `else` acts as a second `if` command that is executed only if the previous test is false. You can have as many `else-then` constructions as you want. As well, the `else` command is optional.

Here is one last version of the script to test for the type of a file: Again, the script has improvements, one of which is that it tells you if you have forgotten to supply an argument when you executed the script.

```bash
# filetype: shell script to determine the type of a file
# arguments:
#   $1 -- the name of a file
# display a message saying if the file is an ordinary file, a directory, or neither
if [ $1 ]
  then if [ -f $1 ]
      then echo "$1 is an ordinary file."
      elif [ -d $1 ]
      then echo "$1 is a directory"
      else echo "$1 is not an ordinary file or directory."
      fi
  else echo "You forgot to specify an argument."
  fi
```

In this example, there are two `else` commands. When the shell encounters the first, it has no way of knowing to which of the previous `elif` or `if` commands the `else` belongs. (Remember, the shell ignores your indentation.) Thus, to avoid any ambiguity, the shell assumes that an `else` command always belongs to the previous `elif` or `if` command.
The case Command

The case command allows you to examine a character string and choose a course of action based on the value of that string. For example, a string may take on one of several different values. Each value requires that a different list of commands be executed. Although you can do this with a sequence of if and elif commands, the case command is more compact and easier to understand.

The syntax is:

```
case string in
  pattern [ | pattern]... ) commands;;
  esac
```

where string stands for a character string; pattern stands for a regular expression that specifies a pattern; and commands stand for one or more commands. You can have as many lines with patterns and commands as you want.

Here is an example of the case command:

```
case $choice in
  time ) echo "The time is 'date'.";;
  dir ) echo "Your working directory is 'pwd'.";;
  esac
```

The case command works as follows: The shell compares the string to the patterns, one by one, starting with the first. When it finds a pattern that matches the string, it executes the corresponding commands. If no pattern matches, no commands are executed.

In the example above, the shell compares the value of the variable choice to the string time. If they are the same, a command is executed that displays the time. If not, the shell compares the variable to the string dir. If they are the same, a command is executed to display the name of the working directory. If not, there is no match and no commands are executed.

You can specify more than one pattern on a line by separating them by vertical bar (|) characters. In this case, the corresponding commands are executed if the string matches either of the patterns. We can change the last example, for instance, to display the time if the variable choice has either of the values time or date:

```
case $choice in
  time | date ) echo "The time is 'date'.";;
  dir ) echo "Your working directory is 'pwd'.";;
  esac
```

When you specify a pattern, you can use a regular expression that
follows the same rules as when you specify file names. (See Chapter 9.)

* matches zero or more characters
? matches any one character
[ ] matches one of the enclosed characters
[!] matches any character that is not enclosed

Inside [ and ], you can use - to indicate a range of single letters or digits.

Here is an example of a shell script that uses regular expressions to match patterns within a case command. The purpose of this script is to search for a file. You can search for one of your own files or search for a file in the directories where the Xenix commands are kept. This latter search is useful when you are creating a script and you want to make sure that the name you choose is not already the name of a command.

The script is called locate. It requires two arguments. The first argument is an option. The second argument is the name of the file for which you want to search. There are three options:

- `u` search user's directories
- `c` search Xenix command directories
- `s` search Xenix command directories

(c stands for “command”; s stands for “system.”)

Thus, to search for one of your files named info,

```bash
locate -u info
```

To check to see if enable is the name of a command, enter either of the commands:

```bash
locate -c enable
locate -s enable
```

Here is the script. Notice the use of the pattern *. Since the * character will match anything, this pattern will be matched as a last resort if no other patterns are matched first.

```bash
# locate: shell script to search for a file
# search for the file whose name is given as $2
# if $1 is -u, search the user's directories
# if $1 is -c or -s, search the Xenix command directories
case $1 in
  -u ) echo "Searching /usr/'logname' for: $2"
        find /usr/'logname' -name $2 -print;;
  -[cs] ) echo "Searching for command: $2"
        find /bin /usr/bin /etc -name $2 -print;;
  * ) echo 'Invalid first argument.';;
esac
# finished
```
The **while** Command

The **while** command allows you to specify that a sequence of commands is to be executed repeatedly, as long as a particular test is true. The syntax is:

```bash
while test
do
  commands
done
```

where *test* is a test of a condition; and *commands* is a sequence of one or more commands.

Here is how the **while** command works. The shell tests the condition that you specify. If the result is true, the shell executes the commands and tests the condition again. If the condition is still true, the shell executes the commands and tests the condition again. This continues until the condition yields a false test. Of course, one of the commands must eventually cause the condition to become false, or the command will loop forever.

The following is an example of a shell script that uses a **while** command. The name of the script is `print.files`. It prints all the files whose names are given as arguments.

Notice how the **while** command creates a loop. The test determines if `$1` is not empty. If not, `$1` is processed. Then, the next argument is shifted into `$1`. When the last argument has been processed, the **shift** command will shift an empty string into `$1`. The test will be false and the loop will terminate.

```bash
# print.files: shell script to print files
# arguments: the names of the files to be printed
# process each argument; if it is the name of an
# ordinary file, print that file
while [ "$1" ]
do
  if [ -f "$1" ];
  then echo "Printing: $1"
    pr $1 | 1pr
  else echo "$1 is not the name of a file"
  fi
  shift  # shift next argument into $1
done  # finished
```
The **while** command can also be used to create continuous loops. If you use the value **true** instead of a test, the **while command** will never stop. Here is an example:

The script is called **snoop**. It loops continuously, waiting for someone to log in or log out. When this happens, **snoop** displays a message. The variable **users** contains the number of users currently logged in. When someone logs in or out, the value of **users** will change.

```bash
# snoop: shell file to detect when someone logs in or out
# save the number of current users
users='who | wc -1'

# loop forever; if someone logs in or out, display a message
while true
do
  if [ $users = 'who | wc -1' ]
    then
      echo '*** Someone has logged in or out ***'
      users='who | wc -1' # reset number of users
  fi
done

# finished
```

If you think about it, you will see that this script is really a daemon. To keep from slowing down the system, you should run **snoop** in the background with a very low preference. Use the **nice** command as follows:

```bash
nice -39 snoop &
```

Make sure to write down the processid. This allows you to use the **kill** command to stop the script, if necessary. (The **nice** and **kill** commands are explained in Chapter 14.)

---

**The until Command**

The **until** command is similar to the **while** command. The only difference is that **until** repeats as long as the test is false. The syntax is:

```bash
until test
do
  commands
done
```
where test is a test of a condition, and commands is a sequence of one or more commands.

Here is a shell script that uses the repeat command. The script is called wait.for. You pass it one argument that is a userid. The script waits for that userid to log in and then sends you a message.

The script works by using grep to extract all the lines from the output of the who command that contain the specified userid. These lines are then counted by lc. If the userid is logged in, its name will appear in the output of who, and grep will extract at least one line. Thus, if the userid in $1 is logged in, the quantity:

$$\text{who} \mid \text{grep } \$1 \mid \text{wc} -1$$

will be greater than zero. If the userid is not logged in, this quantity will be zero. The repeat command tests this quantity and repeats until it is greater than zero; that is, until the userid logs in.

```
# wait.for: shell script to wait for someone to log in
#
# arguments:
# $1 -- the userid you want to wait for
#
# variables:
# login -- the number of lines in the output of the who command, that contain $1
#
# check if the userid is already logged in;
# if not, wait until it is
# login=`who \mid grep $1 \mid wc -1`
if [ $login -gt 0 ]
    then echo "$1 is already logged in."
else
    until [ $login -gt 0 ]
        do
            login=`who \mid grep $1 \mid wc -1`
        done
    done
echo "$1 has logged in"
```

Like the script snoop in the last section, wait.for is a daemon. You should execute it in the background with a very low preference. For example, if you want to be notified when userid sam logs in,

```bash
enter: nice -39 wait.for sam &
```
The for Command

The for command uses a variable name, a list of strings and a sequence of commands. The variable is assigned the value of each string in the list, one at a time. The commands are executed once for every value that the variable takes on. The syntax is:

```
for var in list
do
    commands
done
```

where var is the name of a variable; list is a list of character strings; and commands is a sequence of commands.

Here is an example:

```
for word in Hello to you
do
    echo $word
done
```

This for command assigns the variable word, the values Hello, to, and you. Each time, the echo command is executed. The output is:

```
Hello
to
you
```

Here is an example of a shell script, called `weekly.print`, that uses a for command. This script prints the files named `monday`, `tuesday`, `wednesday`, `thursday`, and `friday` in the directory `usr/acme/sales/reports`.

```
# weekly.report: shell script to print reports for one week
# files: the reports are in /usr/acme/sales/reports,
#       under the names of the days of the week
# for each day of the week, print the report
# for day in mon tues wednes thurs fri
do
    pr /usr/acme/sales/reports/${day}day | lpr
done
```

When you specify the list of strings, you can use a regular expression to refer to file names. For example, to display all the files in the working directory, use:
for file in *
  do
    more $file
  done

Here is a different version of the script `weekly.report` that prints all the files that have names that end in `day`:

```bash
# weekly.report: shell script to print reports for one week
# files: the reports are in /usr/acme/sales/reports,
#       under the names of the days of the week
# for each day of the week, print the report
  for filename in /usr/acme/sales/reports/*day
    do
      more $filename
    done
# finished
```

There is a second form of the `for` command that has the following syntax:

```bash
for var
  do
    commands
  done
```

Notice that the list has been omitted. In this case, the variable takes on the values of the positional parameters, one by one. All the parameters are used, even if there are more than nine. If there are no positional parameters, the commands are not executed.

Here is an example of a script named `display.files` that illustrates this feature. This script displays files from a particular directory. The first positional parameter contains the directory name; the rest of the parameters contain the file names. For example, to display the files mentioned above, you would use:

```bash
display.files /usr/acme/sales/reports monday tuesday\
               wednesday thursday friday
```
Here is the script:

```bash
# display.files: shell script to display files
# arguments:
#   $1 -- the name of the directory in which the files reside
#   $2... -- the names of the files to be displayed

# save the name of the directory
dir=$1

# shift the positional parameters, so that only the file names remain
shift

# process each remaining positional parameter; if it is a file name, display the file
for name
  do
    if [ -f ${dir}/$name ];
      then more ${dir}/$name
    else echo "Invalid file name: ${dir}/$name"
    fi
  done

# finished
```

### The exit Command

The `exit` command stops the execution of a shell script. You can use this command to stop a script when an error has been detected. The syntax is:

```
exit
```

Here is an example of a script that shows how to use the `exit` command. The script is an improved version of `display.files` from the previous section. The old version checks to make sure that the file names are valid. This version also checks that the user supplied at least two arguments and that the first argument is a valid directory name.
# display.files: shell script to display files

# arguments:
#   $1 -- the name of the directory in which the files reside
#   $2... -- the names of the files to be displayed

# make sure there are at least two arguments
if [ $# -lt 2 ];
then echo 'You must supply at least two arguments.'
   exit
fi

# make sure the directory name is valid
if [ ! -d $1 ];
then echo "Invalid directory name: $1"
   exit
fi

# save the name of the directory
dir=$1

# shift the positional parameters, so that only the file names remain
shift

# process each remaining positional parameter; If it is a file name, display the file
for name
   do
      if [ -f ${dir}/$name ];
         then more ${dir}/$name
      else echo "Invalid file name: ${dir}/$name"
         fi
   done

# finished

SETTING UP YOUR WORKING ENVIRONMENT

The .profile File

Each time you log in, the shell looks for a file named .profile in your home directory. For instance, if your userid is sam, the shell looks for
the file /usr/sam/.profile. The shell then executes this file as a shell script. This is done before you enter commands from the keyboard.

When the system manager creates a new userid, Xenix automatically makes a .profile file in the new home directory. If you want, you can modify your .profile file by using the vi editor.

The .profile file has two main uses. First, it contains any commands that you want to execute each time you log in. In this way, you can customize your working environment. For example, you might include the command:

```
  echo 'Hi there good-looking'
```

to greet you every time you log in.

Some people use this feature to automate a task. For example, say that you have an elaborate backup system that is completely run from a shell script. Create a userid called backup and modify the .profile file so it executes the shell script that runs the backup. Now, every time userid backup logs in, a backup is automatically made. This can be done every day by an untrained person, freeing the system manager to work on other things.

The second use of the .profile file is to set and export global variables that will be available to every command and script that you use. There are a few such variables that Xenix sets and uses. They are already in your .profile file, and you can change them if you want.

Three of these are useful for you to know about: HOME, TERM, and PATH. In keeping with convention, the names are in upper case, to help you remember that they are global variables.

**Your Home Directory**

The HOME variable contains the name of your home directory. This is the directory that becomes your working directory every time you log in. It is also the directory that the cd command uses when you don't supply an argument.

Usually, you will not want to change the value of HOME. The reason it is useful is that it is accessible at all times as the name of your home directory. For example, say that your home directory is /usr/dianne. Your working directory is /usr/dianne/projects/winter, and you want to change to /usr/dianne/save. You could enter either of the commands:

```
  cd /usr/dianne/save
  cd ../../save
```
but it is easier to enter:

```
cd $HOME/save
```

When you design a shell file, it is often convenient to refer to a user’s home directory as `$HOME`; this will work for any user at any time, and you do not have to know the userid.

**Setting Your Terminal Type**

In Chapter 13, I explained that Xenix needs to know what type of terminal you are using. If you are using the console, you do not need to do anything. But if you are using a remote terminal, you must set the global variable `TERM` to the name of your type of terminal. Make sure to export the variable.

Usually, you do this in your `.profile` file. You will have to decide what type of terminal you will be using most of the time and add the appropriate commands. For example, if you use an IBM 3101 terminal, add the following commands to your `.profile` file:

```
TERM=3101; export TERM
```

From now on, whenever you log in, Xenix will expect you to be using an IBM 3101 terminal. If, on one occasion, you log in from a different terminal, you can redefine the value of `TERM` before you start. For example, if you on one day use a VT100 terminal, you would enter:

```
TERM=vt100; export TERM
```

from the keyboard as your first two commands.

The names of the different terminals that Xenix can use are listed in the file `/etc/termcap`. (See Chapter 13 for more details.) You can search this file for a particular name by using the `grep` program. Note: The names use lower-case letters.

**Setting the Search Path**

Whenever you enter a command, the shell looks for a program of that name to execute. For example, when you enter:

```
date
```

the shell searches for and executes a program named `date`. If you were to enter:
the shell would search, unsuccessfully, for a program by that name, and then display:

```
  datx: not found
```

The question is, How does the shell know where to look for programs? The answer is, The global variable `PATH` contains a SEARCH PATH that tells the shell where to look.

A search path is a list of directories, separated by colons. Every time you enter a command, the shell looks in each directory that is specified in the search path. If you want to see the search path that the shell is using for your userid, display your `.profile` file by entering:

```
  more $HOME/.profile
```

You will see the line that sets the global variable `PATH`. Unless you have changed it, it will be:

```
  PATH=/bin:/usr/bin:$HOME/bin:. # set command search path
```

This means that the shell will look in the following directories for a program: `/bin`, `/usr/bin`, `$HOME/bin`, and your working directory, whatever it happens to be at the time. (Remember, the single period after the last colon means your working directory.)

The directories that hold commands are named `bin`, because many of the programs are binary files (see Chapter 7). If you create your own shell scripts or binary files, the convention is to put them in your own `bin` directory.

Here is how it works: When you are testing a new shell script, you can execute it just by entering its name. Since the search path contains an entry for the working directory, the shell will find and execute the file.

However, once the shell script is finished, you will want to be able to execute it even if you change your working directory. To do this, move the script to your personal `bin` directory.

For example, say that your userid is `sam`. Create a directory named `/usr/sam/bin`. When you are finished creating a new script, use the `mv` command to move the script to this directory. Since the search path contains the entry `$HOME/bin`, the shell will look in `/usr/sam/bin` and find the script.

The shell searches for programs in the order specified in the search path. Since almost all of the programs that you use are in the directories `/bin` and `/usr/bin`, they are put first. This way, they are searched first.
If you want, you can change the command in the `.profile` file that sets the search path to search your own directories first. For example:

```
PATH=.:/$HOME/bin:/bin/usr/bin
```

This way, you can create a script with the same name as a Xenix command, and the shell will find your script first. However, this is not recommended. Aside from the confusion it could cause having duplicate names, the shell will have to search two extra directories every time you enter a Xenix command, and things will be slower.

Occasionally, you may want to execute a program in a directory that is not in the search path. In this case, specify the directory name in front of the program name. The shell will look in that directory rather than in the usual directories.

For example, the `/etc` directory contains programs that are usually only used by the system manager. (As a matter of fact, the search path for the superuser contains the name `/etc`.) If you want to use one of these programs, you need to preface the name with `/etc`. For example,

```
/etc/mount
/etc/umount
/etc/mkfs
```

(Note: Most of the programs in `/etc` have execute permission only for the superuser.)
APPENDIX A

ASCII CONVERSION TABLE
<table>
<thead>
<tr>
<th>Decimal</th>
<th>Octal</th>
<th>Hex</th>
<th>Binary</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>000</td>
<td>00</td>
<td>0000 0000</td>
<td>(nul)</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>01</td>
<td>0000 0001</td>
<td>(soh)</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>02</td>
<td>0000 0010</td>
<td>(stx)</td>
</tr>
<tr>
<td>003</td>
<td>003</td>
<td>03</td>
<td>0000 0011</td>
<td>(etx)</td>
</tr>
<tr>
<td>004</td>
<td>004</td>
<td>04</td>
<td>0000 0100</td>
<td>(eot)</td>
</tr>
<tr>
<td>005</td>
<td>005</td>
<td>05</td>
<td>0000 0101</td>
<td>(enq)</td>
</tr>
<tr>
<td>006</td>
<td>006</td>
<td>06</td>
<td>0000 0110</td>
<td>(ack)</td>
</tr>
<tr>
<td>007</td>
<td>007</td>
<td>07</td>
<td>0000 0111</td>
<td>(bel)</td>
</tr>
<tr>
<td>008</td>
<td>010</td>
<td>08</td>
<td>0000 1000</td>
<td>(bs)</td>
</tr>
<tr>
<td>009</td>
<td>011</td>
<td>09</td>
<td>0000 1001</td>
<td>(ht)</td>
</tr>
<tr>
<td>010</td>
<td>012</td>
<td>0A</td>
<td>0000 1010</td>
<td>(nl)</td>
</tr>
<tr>
<td>011</td>
<td>013</td>
<td>0B</td>
<td>0000 1011</td>
<td>(vt)</td>
</tr>
<tr>
<td>012</td>
<td>014</td>
<td>0C</td>
<td>0000 1100</td>
<td>(np)</td>
</tr>
<tr>
<td>013</td>
<td>015</td>
<td>0D</td>
<td>0000 1101</td>
<td>(cr)</td>
</tr>
<tr>
<td>014</td>
<td>016</td>
<td>0E</td>
<td>0000 1110</td>
<td>(so)</td>
</tr>
<tr>
<td>015</td>
<td>017</td>
<td>0F</td>
<td>0000 1111</td>
<td>(si)</td>
</tr>
<tr>
<td>016</td>
<td>020</td>
<td>10</td>
<td>0001 0000</td>
<td>(dle)</td>
</tr>
<tr>
<td>017</td>
<td>021</td>
<td>11</td>
<td>0001 0001</td>
<td>(dc1)</td>
</tr>
<tr>
<td>018</td>
<td>022</td>
<td>12</td>
<td>0001 0010</td>
<td>(dc2)</td>
</tr>
<tr>
<td>019</td>
<td>023</td>
<td>13</td>
<td>0001 0011</td>
<td>(dc3)</td>
</tr>
<tr>
<td>020</td>
<td>024</td>
<td>14</td>
<td>0001 0100</td>
<td>(dc4)</td>
</tr>
<tr>
<td>021</td>
<td>025</td>
<td>15</td>
<td>0001 0101</td>
<td>(nak)</td>
</tr>
<tr>
<td>022</td>
<td>026</td>
<td>16</td>
<td>0001 0110</td>
<td>(syn)</td>
</tr>
<tr>
<td>023</td>
<td>027</td>
<td>17</td>
<td>0001 0111</td>
<td>(etb)</td>
</tr>
<tr>
<td>024</td>
<td>030</td>
<td>18</td>
<td>0001 1000</td>
<td>(can)</td>
</tr>
<tr>
<td>025</td>
<td>031</td>
<td>19</td>
<td>0001 1001</td>
<td>(em)</td>
</tr>
<tr>
<td>026</td>
<td>032</td>
<td>1A</td>
<td>0001 1010</td>
<td>(sub)</td>
</tr>
<tr>
<td>027</td>
<td>033</td>
<td>1B</td>
<td>0001 1011</td>
<td>(esc)</td>
</tr>
<tr>
<td>028</td>
<td>034</td>
<td>1C</td>
<td>0001 1100</td>
<td>(fs)</td>
</tr>
<tr>
<td>029</td>
<td>035</td>
<td>1D</td>
<td>0001 1101</td>
<td>(gs)</td>
</tr>
<tr>
<td>030</td>
<td>036</td>
<td>1E</td>
<td>0001 1110</td>
<td>(rs)</td>
</tr>
<tr>
<td>031</td>
<td>037</td>
<td>1F</td>
<td>0001 1111</td>
<td>(us)</td>
</tr>
<tr>
<td>Decimal</td>
<td>Octal</td>
<td>Hex</td>
<td>Binary</td>
<td>Character</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>-----</td>
<td>----------</td>
<td>-----------</td>
</tr>
<tr>
<td>032</td>
<td>040</td>
<td>20</td>
<td>0010 0000</td>
<td>(space)</td>
</tr>
<tr>
<td>033</td>
<td>041</td>
<td>21</td>
<td>0010 0001</td>
<td>!</td>
</tr>
<tr>
<td>034</td>
<td>042</td>
<td>22</td>
<td>0010 0010</td>
<td>&quot;</td>
</tr>
<tr>
<td>035</td>
<td>043</td>
<td>23</td>
<td>0010 0011</td>
<td>#</td>
</tr>
<tr>
<td>036</td>
<td>044</td>
<td>24</td>
<td>0010 0100</td>
<td>$</td>
</tr>
<tr>
<td>037</td>
<td>045</td>
<td>25</td>
<td>0010 0101</td>
<td>%</td>
</tr>
<tr>
<td>038</td>
<td>046</td>
<td>26</td>
<td>0010 0110</td>
<td>&amp;</td>
</tr>
<tr>
<td>039</td>
<td>047</td>
<td>27</td>
<td>0010 0111</td>
<td>,</td>
</tr>
<tr>
<td>040</td>
<td>050</td>
<td>28</td>
<td>0010 1000</td>
<td>(</td>
</tr>
<tr>
<td>041</td>
<td>051</td>
<td>29</td>
<td>0010 1001</td>
<td>)</td>
</tr>
<tr>
<td>042</td>
<td>052</td>
<td>2A</td>
<td>0010 1010</td>
<td>*</td>
</tr>
<tr>
<td>043</td>
<td>053</td>
<td>2B</td>
<td>0010 1011</td>
<td>+</td>
</tr>
<tr>
<td>044</td>
<td>054</td>
<td>2C</td>
<td>0010 1100</td>
<td>,</td>
</tr>
<tr>
<td>045</td>
<td>055</td>
<td>2D</td>
<td>0010 1101</td>
<td>-</td>
</tr>
<tr>
<td>046</td>
<td>056</td>
<td>2E</td>
<td>0010 1110</td>
<td>.</td>
</tr>
<tr>
<td>047</td>
<td>057</td>
<td>2F</td>
<td>0010 1111</td>
<td>/</td>
</tr>
<tr>
<td>048</td>
<td>060</td>
<td>30</td>
<td>0011 0000</td>
<td>0</td>
</tr>
<tr>
<td>049</td>
<td>061</td>
<td>31</td>
<td>0011 0001</td>
<td>1</td>
</tr>
<tr>
<td>050</td>
<td>062</td>
<td>32</td>
<td>0011 0010</td>
<td>2</td>
</tr>
<tr>
<td>051</td>
<td>063</td>
<td>33</td>
<td>0011 0011</td>
<td>3</td>
</tr>
<tr>
<td>052</td>
<td>064</td>
<td>34</td>
<td>0011 0100</td>
<td>4</td>
</tr>
<tr>
<td>053</td>
<td>065</td>
<td>35</td>
<td>0011 0101</td>
<td>5</td>
</tr>
<tr>
<td>054</td>
<td>066</td>
<td>36</td>
<td>0011 0110</td>
<td>6</td>
</tr>
<tr>
<td>055</td>
<td>067</td>
<td>37</td>
<td>0011 0111</td>
<td>7</td>
</tr>
<tr>
<td>056</td>
<td>070</td>
<td>38</td>
<td>0011 1000</td>
<td>8</td>
</tr>
<tr>
<td>057</td>
<td>071</td>
<td>39</td>
<td>0011 1001</td>
<td>9</td>
</tr>
<tr>
<td>058</td>
<td>072</td>
<td>3A</td>
<td>0011 1010</td>
<td>:</td>
</tr>
<tr>
<td>059</td>
<td>073</td>
<td>3B</td>
<td>0011 1011</td>
<td>;</td>
</tr>
<tr>
<td>060</td>
<td>074</td>
<td>3C</td>
<td>0011 1100</td>
<td>&lt;</td>
</tr>
<tr>
<td>061</td>
<td>075</td>
<td>3D</td>
<td>0011 1101</td>
<td>=</td>
</tr>
<tr>
<td>062</td>
<td>076</td>
<td>3E</td>
<td>0011 1110</td>
<td>&gt;</td>
</tr>
<tr>
<td>063</td>
<td>077</td>
<td>3F</td>
<td>0011 1111</td>
<td>?</td>
</tr>
<tr>
<td>Decimal</td>
<td>Octal</td>
<td>Hex</td>
<td>Binary</td>
<td>Character</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>-----</td>
<td>-----------</td>
<td>-----------</td>
</tr>
<tr>
<td>064</td>
<td>100</td>
<td>40</td>
<td>0100 0000</td>
<td>@</td>
</tr>
<tr>
<td>065</td>
<td>101</td>
<td>41</td>
<td>0100 0001</td>
<td>A</td>
</tr>
<tr>
<td>066</td>
<td>102</td>
<td>42</td>
<td>0100 0010</td>
<td>B</td>
</tr>
<tr>
<td>067</td>
<td>103</td>
<td>43</td>
<td>0100 0011</td>
<td>C</td>
</tr>
<tr>
<td>068</td>
<td>104</td>
<td>44</td>
<td>0100 0100</td>
<td>D</td>
</tr>
<tr>
<td>069</td>
<td>105</td>
<td>45</td>
<td>0100 0101</td>
<td>E</td>
</tr>
<tr>
<td>070</td>
<td>106</td>
<td>46</td>
<td>0100 0110</td>
<td>F</td>
</tr>
<tr>
<td>071</td>
<td>107</td>
<td>47</td>
<td>0100 0111</td>
<td>G</td>
</tr>
<tr>
<td>072</td>
<td>110</td>
<td>48</td>
<td>0100 1000</td>
<td>H</td>
</tr>
<tr>
<td>073</td>
<td>111</td>
<td>49</td>
<td>0100 1001</td>
<td>I</td>
</tr>
<tr>
<td>074</td>
<td>112</td>
<td>4A</td>
<td>0100 1010</td>
<td>J</td>
</tr>
<tr>
<td>075</td>
<td>113</td>
<td>4B</td>
<td>0100 1011</td>
<td>K</td>
</tr>
<tr>
<td>076</td>
<td>114</td>
<td>4C</td>
<td>0100 1100</td>
<td>L</td>
</tr>
<tr>
<td>077</td>
<td>115</td>
<td>4D</td>
<td>0100 1101</td>
<td>M</td>
</tr>
<tr>
<td>078</td>
<td>116</td>
<td>4E</td>
<td>0100 1110</td>
<td>N</td>
</tr>
<tr>
<td>079</td>
<td>117</td>
<td>4F</td>
<td>0100 1111</td>
<td>O</td>
</tr>
<tr>
<td>080</td>
<td>120</td>
<td>50</td>
<td>0101 0000</td>
<td>P</td>
</tr>
<tr>
<td>081</td>
<td>121</td>
<td>51</td>
<td>0101 0001</td>
<td>Q</td>
</tr>
<tr>
<td>082</td>
<td>122</td>
<td>52</td>
<td>0101 0010</td>
<td>R</td>
</tr>
<tr>
<td>083</td>
<td>123</td>
<td>53</td>
<td>0101 0011</td>
<td>S</td>
</tr>
<tr>
<td>084</td>
<td>124</td>
<td>54</td>
<td>0101 0100</td>
<td>T</td>
</tr>
<tr>
<td>085</td>
<td>125</td>
<td>55</td>
<td>0101 0101</td>
<td>U</td>
</tr>
<tr>
<td>086</td>
<td>126</td>
<td>56</td>
<td>0101 0110</td>
<td>V</td>
</tr>
<tr>
<td>087</td>
<td>127</td>
<td>57</td>
<td>0101 0111</td>
<td>W</td>
</tr>
<tr>
<td>088</td>
<td>130</td>
<td>58</td>
<td>0101 1000</td>
<td>X</td>
</tr>
<tr>
<td>089</td>
<td>131</td>
<td>59</td>
<td>0101 1001</td>
<td>Y</td>
</tr>
<tr>
<td>090</td>
<td>132</td>
<td>5A</td>
<td>0101 1010</td>
<td>Z</td>
</tr>
<tr>
<td>091</td>
<td>133</td>
<td>5B</td>
<td>0101 1011</td>
<td>[</td>
</tr>
<tr>
<td>092</td>
<td>134</td>
<td>5C</td>
<td>0101 1100</td>
<td>\</td>
</tr>
<tr>
<td>093</td>
<td>135</td>
<td>5D</td>
<td>0101 1101</td>
<td>]</td>
</tr>
<tr>
<td>094</td>
<td>136</td>
<td>5E</td>
<td>0101 1110</td>
<td>^</td>
</tr>
<tr>
<td>095</td>
<td>137</td>
<td>5F</td>
<td>0101 1111</td>
<td>_</td>
</tr>
<tr>
<td>Decimal</td>
<td>Octal</td>
<td>Hex</td>
<td>Binary</td>
<td>Character</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>-----</td>
<td>----------</td>
<td>-----------</td>
</tr>
<tr>
<td>096</td>
<td>140</td>
<td>60</td>
<td>0110 0000</td>
<td>'</td>
</tr>
<tr>
<td>097</td>
<td>141</td>
<td>61</td>
<td>0110 0001</td>
<td>a</td>
</tr>
<tr>
<td>098</td>
<td>142</td>
<td>62</td>
<td>0110 0010</td>
<td>b</td>
</tr>
<tr>
<td>099</td>
<td>143</td>
<td>63</td>
<td>0110 0011</td>
<td>c</td>
</tr>
<tr>
<td>100</td>
<td>144</td>
<td>64</td>
<td>0110 0100</td>
<td>d</td>
</tr>
<tr>
<td>101</td>
<td>145</td>
<td>65</td>
<td>0110 0101</td>
<td>e</td>
</tr>
<tr>
<td>102</td>
<td>146</td>
<td>66</td>
<td>0110 0110</td>
<td>f</td>
</tr>
<tr>
<td>103</td>
<td>147</td>
<td>67</td>
<td>0110 0111</td>
<td>g</td>
</tr>
<tr>
<td>104</td>
<td>150</td>
<td>68</td>
<td>0110 1000</td>
<td>h</td>
</tr>
<tr>
<td>105</td>
<td>151</td>
<td>69</td>
<td>0110 1001</td>
<td>i</td>
</tr>
<tr>
<td>106</td>
<td>152</td>
<td>6A</td>
<td>0110 1010</td>
<td>j</td>
</tr>
<tr>
<td>107</td>
<td>153</td>
<td>6B</td>
<td>0110 1011</td>
<td>k</td>
</tr>
<tr>
<td>108</td>
<td>154</td>
<td>6C</td>
<td>0110 1100</td>
<td>l</td>
</tr>
<tr>
<td>109</td>
<td>155</td>
<td>6D</td>
<td>0110 1101</td>
<td>m</td>
</tr>
<tr>
<td>110</td>
<td>156</td>
<td>6E</td>
<td>0110 1110</td>
<td>n</td>
</tr>
<tr>
<td>111</td>
<td>157</td>
<td>6F</td>
<td>0110 1111</td>
<td>o</td>
</tr>
<tr>
<td>112</td>
<td>160</td>
<td>70</td>
<td>0111 0000</td>
<td>p</td>
</tr>
<tr>
<td>113</td>
<td>161</td>
<td>71</td>
<td>0111 0001</td>
<td>q</td>
</tr>
<tr>
<td>114</td>
<td>162</td>
<td>72</td>
<td>0111 0010</td>
<td>r</td>
</tr>
<tr>
<td>115</td>
<td>163</td>
<td>73</td>
<td>0111 0011</td>
<td>s</td>
</tr>
<tr>
<td>116</td>
<td>164</td>
<td>74</td>
<td>0111 0100</td>
<td>t</td>
</tr>
<tr>
<td>117</td>
<td>165</td>
<td>75</td>
<td>0111 0101</td>
<td>u</td>
</tr>
<tr>
<td>118</td>
<td>166</td>
<td>76</td>
<td>0111 0110</td>
<td>v</td>
</tr>
<tr>
<td>119</td>
<td>167</td>
<td>77</td>
<td>0111 0111</td>
<td>w</td>
</tr>
<tr>
<td>120</td>
<td>170</td>
<td>78</td>
<td>0111 1000</td>
<td>x</td>
</tr>
<tr>
<td>121</td>
<td>171</td>
<td>79</td>
<td>0111 1001</td>
<td>y</td>
</tr>
<tr>
<td>122</td>
<td>172</td>
<td>7A</td>
<td>0111 1010</td>
<td>z</td>
</tr>
<tr>
<td>123</td>
<td>173</td>
<td>7B</td>
<td>0111 1011</td>
<td>{</td>
</tr>
<tr>
<td>124</td>
<td>174</td>
<td>7C</td>
<td>0111 1100</td>
<td></td>
</tr>
<tr>
<td>125</td>
<td>175</td>
<td>7D</td>
<td>0111 1101</td>
<td>}</td>
</tr>
<tr>
<td>126</td>
<td>176</td>
<td>7E</td>
<td>0111 1110</td>
<td>~</td>
</tr>
<tr>
<td>127</td>
<td>177</td>
<td>7F</td>
<td>0111 1111</td>
<td>(del)</td>
</tr>
</tbody>
</table>
SUMMARY OF vi COMMANDS
The following arguments are used in this summary:

- **buf**: the name of a delete buffer \([1-9]\) or a named buffer \([a-z]\)
- **BUF**: the upper-case name of a named buffer \([A-Z]\)
- **char**: a character
- **command**: a Xenix command
- **excom**: one or more \texttt{ex} commands, separated by \texttt{I}
- **file**: the name of a text file
- **lchar**: a lower-case letter
- **line**: a line number
- **long**: a string to be abbreviated
- **mark**: the name of a marker set by an \texttt{m} command
- **move**: a cursor movement command that does not use the \(<\text{CTRL}>\)
  or cursor-control keys
- **n**: a number
- **pattern**: a regular expression to be used as a search pattern
- **replace**: a replacement string
- **short**: a string to be used as an abbreviation
- **tagname**: an entry in a tag file
- **target**: a line number after which an insertion is to take place

**Starting \texttt{vi}**

\texttt{vi [-rR] [file...]}

options: \texttt{r}—recover; \texttt{R}—read only

**Stopping \texttt{vi} and Saving Your Work**

- **ZZ**: save editing buffer to current file and stop
- **:x [file]**: save editing buffer and stop
- **:w [file]**: save editing buffer; do not stop
- **:q [file]**: stop without saving editing buffer
- **:wl [file]**: same as \texttt{:w} but override check
- **:q! [file]**: same as \texttt{:q} but override check
Editing More Than One File

:n edit next file in argument list
<CTRL-g> display name of current file
:f same as CTRL-g
:args display argument list
:rew edit first file in argument list
:n [file...] edit new file; change argument list
:e [file] edit new file; same argument list
:rew! same as :rew but override check
:n! same as :n but override check
:e! same as :e but override check

Using ex Commands

When you use ex commands from vi, precede them with a colon. Most ex commands allow you to specify line numbers before the command name. You can use:

*line* one line number
*line,line* range of line numbers

If you do not specify line numbers, most commands use the current line. The argument *line* can be either the actual number of a line or:

. current line
0 beginning of editing buffer
$ last line of editing buffer
% entire file (same as 1,$)
/pattern/ search forward: next line containing *pattern*
/?pattern? search backward: next line containing *pattern*
" line of previous absolute move command
'mark line containing *mark*

To indicate a line before or after a specific line:

-n n lines before
+n n lines after

To change to and from ex, use:

Q change from vi to ex
:vi change from ex to vi
Moving the Cursor

- **G**
  - `lineG` go to the line with the specified number
  - `<ENTER>` go to the last line

- **H** move cursor to top line of screen
- **M** move cursor to middle line of screen
- **L** move cursor to bottom line of screen
- `{` move cursor forward to beginning of paragraph
- `}` move cursor backward to beginning of paragraph
- `)` move cursor forward to beginning of sentence
- `( move cursor backward to beginning of sentence
- **w** move cursor forward to beginning of next word
- **b** move cursor backward to beginning of word
- **W** same as `w`; ignore punctuation
- **E** same as `e`; ignore punctuation
- **B** same as `b`; ignore punctuation
- `^` move cursor to beginning of current line
- `$` move cursor to first non-space/tab in current line
- `nl` move cursor to end of current line
- **tchar** move cursor right to next occurrence of character
- **Fchar** move cursor left to next occurrence of character
- **tchar** move cursor right to position before character
- **Tchar** move cursor left to position before character
- `;` same direction: repeat last `t, F, t, T` command
- `,` reverse direction: repeat last `t, F, t, T` command
- `+` move cursor to first non-space/tab of previous line
- `<ENTER>` move cursor to first non-space/tab of next line
- `<CURSOR-RIGHT>` move cursor one position right
- `<CURSOR-LEFT>` move cursor one position left
- `<CURSOR-UP>` move cursor one position up
- `<CURSOR-DOWN>` move cursor one position down
- `<SPACE>` same as `<CURSOR-RIGHT>`
- `<BACKSPACE>` same as `<CURSOR-LEFT>`
- `<CTRL-p>` same as `<CURSOR-UP>`
- `<CTRL-n>` same as `<CURSOR-DOWN>`
- `l` same as `<CURSOR-RIGHT>`
- `h` same as `<CURSOR-LEFT>`
- `k` same as `<CURSOR-UP>`
- `j` same as `<CURSOR-DOWN>`
Specifying a Search Pattern

The following characters have special meanings within search patterns:

- matches any one character except a newline
- matches zero or more of the preceding characters
^ matches the beginning of a line
$ matches the end of a line
\(<\ matches the beginning of a word
\(>\ matches the end of a word
[ ] matches one of the enclosed characters
[^ ] matches any character that is not enclosed
\( \) define part of a search pattern

Inside [ and ], you can use - to indicate a range of single letters or digits. \ indicates that the following character is to be taken literally, except for \(<, \>, \(, and \).

The following characters have special meanings within replace patterns:

\& last pattern matched
\n use nth part of search pattern defined with \( ( and \)
\u convert next character to upper case
\l convert next character to lower case
\U convert following characters to upper case
\L convert following characters to lower case
\E stop converting to upper or lower case

Searching for Patterns

/pattern/ search forward: go to next line with pattern
?pattern/ search backward: go to next line with pattern
/p/ search forward: for previous pattern
?/ search backward: for previous pattern
n same direction: repeat last / or ? command
N reverse direction: repeat last / or ? command
/pattern/+n search forward: go to nth line after pattern
/pattern/-n search forward: go to nth line before pattern
?pattern?+n search backward: go to nth line after pattern
?pattern?–n search backward: go to nth line before pattern
Redrawing the Screen

\[z\langle \text{ENTER}\rangle\]  
redraw screen: current line at top

\[z.\]  
redraw screen: current line at middle

\[z-\]  
redraw screen: current line at bottom

\[linez\langle \text{ENTER}\rangle\]  
redraw screen: specified line at top

\[linez.\]  
redraw screen: specified line at middle

\[linez-\]  
redraw screen: specified line at bottom

\[/pattern/z\langle \text{ENTER}\rangle\]  
redraw screen: line with pattern at top

\[/pattern/z.\langle \text{ENTER}\rangle\]  
redraw screen: line with pattern at middle

\[/pattern/z-\langle \text{ENTER}\rangle\]  
redraw screen: line with pattern at bottom

\?[pattern?z\langle \text{ENTER}\rangle\]  
redraw screen: line with pattern at top

\?[pattern?z.\langle \text{ENTER}\rangle\]  
redraw screen: line with pattern at middle

\?[pattern?z-\langle \text{ENTER}\rangle\]  
redraw screen: line with pattern at bottom

Returning and Marking

The absolute move commands are \texttt{G,/,?,",",', and '}

\[\"\]  
move cursor to place of last absolute move command

\[\"\]  
same as \texttt{":} but move to first non-\texttt{-space/tab} on line

\[m/\text{char}\]  
mark place in editing buffer with specified name

\[l/\text{char}\]  
move cursor to specified place

\[l/\text{char}\]  
same as \texttt{":} but move to first non-\texttt{-space/tab} on line

Undoing and Replacing

\[u\]  
undo last command that modified the editing buffer

\[U\]  
restore current line

\[.\]  
repeat last command that modified the editing buffer

Inserting

\[i\]  
change to insert mode: insert after cursor position

\[I\]  
change to insert mode: insert at end of current line

\[a\]  
change to insert mode: insert before cursor position

\[A\]  
change to insert mode: insert at start of current line

\[o\]  
change to insert mode: open below current line

\[O\]  
change to insert mode: open above current line
Controlling Line Length

:\set\ wr=n \hspace{1em} \text{set automatic margin } n \text{ positions from right}
\ J \hspace{1em} \text{join lines}

Replacing

\ r \hspace{1em} \text{replace exactly one character}
\ R \hspace{1em} \text{replace by typing over}
\ s \hspace{1em} \text{replace one character by insertion}
\ S \hspace{1em} \text{replace current line by insertion}
\ cmove \hspace{1em} \text{replace from cursor to \textit{move} by insertion}
\ C \hspace{1em} \text{replace to end of current line by insertion}
\ cc \hspace{1em} \text{same as } S

All of the above commands, except \texttt{r}, change to insert mode.

:\s/pattern/replace/ \hspace{1em} \text{substitute on current line}
:\lines/pattern/replace/ \hspace{1em} \text{substitute on specified line}
:\line,lines/pattern/replace/ \hspace{1em} \text{substitute over specified range}

Deleting

\ x \hspace{1em} \text{delete character at position of cursor}
\ X \hspace{1em} \text{delete character to left of cursor}
\ dmove \hspace{1em} \text{delete from cursor to \textit{move}}
\ dd \hspace{1em} \text{delete current line}
\ D \hspace{1em} \text{delete from position of cursor to end of line}
:\lined \hspace{1em} \text{delete specified line}
:\line,lined \hspace{1em} \text{delete specified range}

Moving and Copying

:\linetarget \hspace{1em} \text{copy specified line; insert after } target
:\line,linetarget \hspace{1em} \text{copy specified range; insert after } target
:\linemtarget \hspace{1em} \text{move specified line; insert after } target
:\line,linemtarget \hspace{1em} \text{move specified range; insert after } target
Writing and Reading from Files

`:linew file` copy specified line to `file`
`:line,linew file` copy specified range of lines to `file`
`:w>>` same as `:w` but add to file
`:linere file` insert contents of `file` after `line`
`:linere! command` insert Xenix command output after `line`

Executing a Sequence of ex Commands

`:g/pattern/excom` execute `excom` on lines containing `pattern`
`:line,lineg/pattern/excom` same as `:g` over specified range
`:gi/pattern/excom` execute `excom` on lines not containing `pattern`
`:line,linegi/pattern/excom` same as `:gi` over specified range
`:so file` execute commands from `file`

Using Buffers

`p` copy from unnamed buffer; insert after cursor
`P` copy from unnamed buffer; insert before cursor
"bufp` same as `p` but use specified buffer
"bufP` same as `P` but use specified buffer
`ymove` copy to unnamed buffer from cursor position to `move`
`yy` copy one line to unnamed buffer
`Y` same as `yy`
"bufymove` same as `y` but use specified buffer
"bufyy` same as `yy` but use specified buffer
"bufY` same as `Y` but use specified buffer
"BUFymove` same as `y` but add onto specified buffer
"BUFyy` same as `yy` but add onto specified buffer
"BUFy` same as `Y` but add onto specified buffer
"bufdmove` same as `d` but save in specified buffer
"bufdd` same as `dd` but save in specified buffer
"bufD` same as `D` but save in specified buffer
"BUFdmove` same as `d` but save onto end of specified buffer
"BUFdd` same as `dd` but save onto end of specified buffer
"BUFD` same as `D` but save onto end of specified buffer
Using Xenix Commands

```
:command         execute the specified Xenix command
:!!               repeat previous Xenix command
:ish              pause vi and start a new copy of the shell
:linere! command  insert Xenix command output at specified line
:rel command      insert Xenix command output at current line
!move command     execute Xenix command from cursor to move
n!!command        execute Xenix command on n lines
```

Using Abbreviations

```
:ab short long    set short as an abbreviation for long
:ab               display current abbreviations
:uab short        cancel abbreviation short
```

Editing Computer Programs

```
<move              shift left: from current line to move
>move              shift right: from current line to move
<<                 shift left: current line
>>                 shift right: current line
n<<                 shift left: specified number of lines
n>>                 shift right: specified number of lines
[[                  move cursor backward to line that begins with {
]]                 move cursor forward to line that begins with {
:tagname           jump to function named tagname
:ta                jump to previous tagname
```

To start vi using a file that contains a specified tag, use:

```
vi -t tagname
```
GLOSSARY

ABSOLUTE MOVE COMMAND
a command that moves the cursor to a specific line

ARGUMENT
the part of the command that specifies information that the command needs to carry out its task

ASCII CODE
a standardized representation of characters by particular patterns of bits

BACKGROUND
describes a process in which the program that is running cannot read and write to the terminal; such processes are said to be running “in the background”

BACKUP
a copy of information

BACKUP
to make a copy of information

BINARY FILE
an ordinary file in which the bytes are considered to represent numbers in the binary system

BINARY SYSTEM
a system of counting using only the digits 0 and 1

BIT
the smallest component of computer memory

BLIND COPY
a copy of a message sent in such a way that only the sender and recipient know who received the copy

BODY
the second part of a message, which contains the text of the message
BOURNE SHELL
a shell developed by S. R. Bourne at Bell Labs; the most commonly used shell

BUFFER
a temporary holding area for data that is being transferred

BYTE
a sequence of eight bits

C-SHELL
a shell often used by programmers and experienced users; the name comes from the fact that this shell has features in common with the programming language C; pronounced: "see-shell"

COMMAND LINE
the bottom lines of the screen, used by vi to display messages and commands

COMMAND MODE
[vi, mail] a state in which a user can enter commands

COMMAND PROCESSOR
a program that reads and carries out commands that are entered from the keyboard

COMMENT
within a shell script, a descriptive line that is ignored by the shell when the script is executed

COMPOSE MODE
[mail] a state in which a user can type a message

CONSOLE
the name of the terminal consisting of the keyboard and display that is part of the computer on which Xenix resides

CONTROL FLOW
the logic that describes the order in which the instructions in a program are executed

CURRENT FILE
the text file that you are working with at a particular time

CURRENT LINE
the line on which the cursor lies

CURRENT MESSAGE
the last message that was displayed
CURSOR
the blinking underscore symbol that marks the current position on the display screen

CURSOR-CONTROL KEYS
the four keys with arrows on the numeric keypad that change the position of the cursor

DAILY BACKUP
(sysadmin) a backup of a file system that contains only those files that have been added or modified since the previous periodic backup

DATA
information used by a program

DEAMON
a background process started automatically by Xenix to provide a service of general interest

DECIMAL SYSTEM
a system of counting using the digits 0 through 9

DEFAULT
for each option, the value that is automatically assigned when vi starts

DELETE BUFFER
one of nine temporary storage areas, used by vi to hold the nine most recent deletions that involved a line, a sentence, or anything longer; the delete buffers are numbered 1 through 9

DIRECTORY
a file that is used to keep track of other files

DISMOUNT
to disconnect a file system from the main tree

DISPLAY
the television-like device upon which the computer writes information;
to write information upon a display

DOUBLE-SIDED DISK DRIVE
a device that uses diskettes that contain up to 360 kilobytes of data

ECHO
to display characters on the screen as you type them;
to write information exactly as it was read

EDITING BUFFER
a working copy of the current file
EDITING SCRIPT
a text file that contains ex commands

EDITOR
a program that you use to create and modify text files

EMPTY
describes a variable that has not been assigned a value

ENTER
to type information and then press the <ENTER> key

eof
"end of file"; the character produced by pressing <CTRL-d>; eof signals a program that is reading data that there is no more data; <CTRL-d> is often used to stop programs that read data from the keyboard; pronounced: "ee-oh-eff"

erase
the character produced by pressing the <BACKSPACE> key; erase instructs the shell to erase the last character you typed

EXECUTABLE PROGRAM
a program stored in a form that the computer can understand

EXECUTE [a program]
to perform the instructions in a program

EXPONENT
(in scientific notation) the number that indicates the scale of the value being described

EXPORT
within a shell script, to pass the value of a variable to scripts that are subsequently executed

FACTOR
a whole number that evenly divides into another whole number; for example, 12 has six factors: 1, 2, 3, 4, 6, and 12.

FIELD
within a record, one item of information

FILE
a source from which data can be read or a target to which data can be written

FILE SYSTEM
a group of files organized as a tree
FILTER
any program that reads from the standard input and writes to the standard output

FOREGROUND
describes a process in which the program that is running can read and write to the terminal; such processes are said to be running "in the foreground"

FORMAT
to prepare a new diskette

FUNCTION-KEY AREA
the left-hand part of the keyboard, containing the ten keys labeled <F1> through <F10>

GLOBAL VARIABLE
a variable that has been exported; hence, its value is available to more than one shell script

GROUP
a set of userids; each userid is assigned to a group by the system manager

HEADER
a one-line description of a message

HEADING
the first part of a message, which contains general information about the message

HIGH-CAPACITY DISK DRIVE
a device that uses diskettes that contain up to 1200 kilobytes of data

HOME DIRECTORY
the directory to which the shell automatically sets your working directory whenever you log in

INPUT
information supplied to the program while it is executing

INPUT/OUTPUT
the general concept of reading and writing information; abbreviated: I/O

INSERT MODE
the mode in which you can add information to the editing buffer

INSTALL [Xenix]
to set up the Xenix system for use with your computer
INUMBER
"index number"; a unique number, used internally by Xenix to represent a particular file; pronounced: "eye-number"

I/O
abbreviation for INPUT/OUTPUT

intr
"interrupt"; the character produced by pressing the <DEL> key; <DEL> is often used to stop a command or program before it is finished

KEY
the part of a record that is compared during a sorting operation

KEYBOARD BUFFER
the buffer in which Xenix collects the characters that you type at the keyboard

kill
the character produced by pressing <CTRL-u>; kill instructs the shell to erase the entire line that you are typing

LINE EDITORS
an editor that does not make full use of the screen, but rather works with and displays groups of lines

LINK
an entry in a directory for a particular file

LOCAL VARIABLE
a variable that has not been exported; hence, its value is available only within one shell script

LOG IN
to start a work session with Xenix

LOG OUT
to end a work session with Xenix

MENU
a list of choices displayed on the screen

MESSAGE
a text file that is sent from one userid to another by the Xenix mail system

MESSAGE LIST
a specification of one or more messages

MODE
a particular way in which a program can work
MOUNT
to connect a file system to the main tree

NAMED BUFFER
one of 26 temporary storage areas available to the user; the named buffers are
known by the letters of the alphabet, a through z

newline
the character produced by pressing the <ENTER> key

NUMERIC KEYPAD
the right-hand part of the keyboard, containing a variety of special keys

OPERATING SYSTEM
a complex program that coordinates the overall functioning of a computer

OPTION
a value you can set that lets you specify how you want vi to behave;
the part of a command that gives you control over how the command
carries out its task

ORDINARY FILE
a file that contains data

OUTPUT
information generated by a program

OWNER
the userid that controls the access to a particular file

PARENT DIRECTORY
a directory that contains another directory

PASSWORD
a sequence of characters that a user must specify to access the Xenix system;
each userid has its own password

PATH
a sequence of directories that leads through the tree to a particular directory

PATHNAME
the full specification of a file, including the path to the directory in which the
file lies, followed by the name of the file

PERIODIC BACKUP
(sysadmin) a backup of a file system that contains all the files

PERMISSIONS
an attribute of a file that specifies how that file may be accessed
PIPE
the mechanism that connects two adjacent programs within a pipeline; when you create a pipeline, you use the | (vertical bar) character to indicate a pipe; PIPE is also used as a verb, meaning to pass data from one program to another

PIPELINE
an arrangement in which data is passed through a sequence of programs

POSITIONAL PARAMETER
within a shell script, a variable that is assigned the value of one of the arguments that was specified when the script was executed

PRIMARY DISPLAY
the display that is used as the output device for the console

PRIME FACTOR
a factor that is a prime number; for example, of all the factors of 12 (1, 2, 3, 4, 6, and 12), only 2 and 3 are prime factors

PRIME NUMBER
a number that has no factors except 1 and itself; for example, 13 is a prime number while 12 is not

PRINT
to write output on paper using a printer
(Xenix manuals often use this term to refer to writing information on a display)

PRINT QUEUE
a list of files waiting to be printed

PRIORITY
a number from 0 to 39 assigned to a process that indicates the preference that the process should receive as Xenix allots the resources of the computer; the lower the number, the greater the preference

PROCESS
the event of running a program

PROCESSID
a unique identification number, assigned to a process by Xenix; pronounced: "process-eye-dee"

PROGRAM
a set of instructions that can be carried out by a computer

PROMPT
one or more characters displayed by a program to let you know that it is waiting for input
READ [input]
to take in information during the execution of a program

RECORD
within a text file, the largest unit of information holding one or more related items of data

REDIRECT
to reassign the standard input or standard output to a particular file

REGULAR EXPRESSION
a compact way of specifying all the character strings that match a particular pattern

RESTORE
to recover lost information by copying a backup

root
a special userid that affords the user many extraordinary privileges; usually used only by the system manager

ROOT DIRECTORY
the main directory of a tree-structured file system

RUN [a program]
same as EXECUTE

SCIENTIFIC NOTATION
a way of writing very large or very small numbers in a compact manner

SCREEN EDITOR
an editor that makes use of the whole screen to allow you to see and work with more than one line at a time

SCRIPT
a text file that contains a sequence of commands to be read by a program

SCROLL
to display a line of information on the display screen by writing to the bottom of the screen and moving all the other lines up one position

SEARCH PATH
the list of directories that the shell searches when it is looking for a program

SHELL
the Xenix name for a command processor

SHELL FILE
same as SHELL SCRIPT
SHELL PROCEDURE
same as SHELL SCRIPT

SHELL SCRIPT
a file of commands that is to be executed by the shell

SORT
to rearrange the records in a text file, according to a specified ordering, based on the values of one or more fields in each record

SOURCE PROGRAM
a program stored in a form that people can understand

space
the character produced by pressing the <SPACE> key

SPECIAL FILE
a file that represents a physical device

STABLE
describes a sorting program that never changes the order of records that have equal key values

STANDARD INPUT
the file from which Xenix commands read information

STANDARD OUTPUT
the file to which Xenix commands write information

start
the character produced by pressing <CTRL-q>; start instructs the shell to continue with the screen display after you have sent the stop character

stop
the character produced by pressing <CTRL-s>; stop instructs the shell to pause the screen display

STRING
a sequence of characters

STRING OPTION
an option that has the value of a string of letters or numbers

SUB-DIRECTORY
a directory that is contained in another directory

SUPERUSER
a person using the root userid
SWITCH OPTION
an option that has a value of either on or off

SYNTAX
a description of how a command must be entered

SYSTEM MAINTENANCE
the tasks that the system manager carries out to take care of the system

SYSTEM MANAGER
the person who maintains the Xenix system

tab
the character produced by pressing the <TAB> key

TAG FILE
a text file that contains information about particular locations in a group of files

TEE
a filter that reads data from the standard input, makes a copy of the data, and then writes it to the standard output

TERMINAL
the keyboard and display unit with which you communicate with Xenix

TEXT FILE
an ordinary file in which each byte represents one character

TREE-STRUCTURE
a scheme of organizing files in which the hierarchy resembles the outline of a tree

TTY
same as TERMINAL; pronounced: "tee-tee-why"

TYPEWRITER-KEY AREA
the center part of the keyboard, containing keys similar to those on a typewriter

UNIX
an operating system originally developed at Bell Labs

UNNAMED BUFFER
a temporary storage area used by vi to hold the most recent deletion

UNSTABLE
describes a sorting program that sometimes changes the order of records that have equal key values
USER
a person who is using the computer system

USER-DEFINED VARIABLE
a variable that is created within a shell script

USERID
an internal Xenix identification name; Xenix knows each user by userid;
   pronounced: "user-eye-dee"

VARIABLE
a quantity with a name that can take on different values

VISUAL SHELL
a shell that was designed to provide a visually pleasing environment; the visual
   shell gives inexperienced and casual users an easy-to-use interface to
   Xenix

WINDOW
a box drawn on the screen in which information is displayed

WORKING DIRECTORY
the name of a particular directory from which you can specify paths

WRITE [output]
to generate information during the execution of a program

XENIX
an operating system based on UNIX; Xenix was developed by Microsoft
   Corporation for use with personal computers
QUICK INDEX OF XENIX COMMANDS

banner, 48  
bc, 56  
cal, 46  
case, 326  
cat, 213  
chgrp, 265  
chmod, 263  
chown, 265  
comm, 251  
copy, 103  
chown, 265  
cp, 102  
date, 45  
diff, 255  
doscat, 296  
doscp, 295  
dosdir, 297  
dosis, 297  
dosmkdir, 297  
dosrm, 296  
dosrmdir, 297  
echo, 47  
egrep, 236  
exit, 333  
export, 313  
factor, 53  
fgrep, 239  
file, 113  
find, 111  
finger, 43  
for, 331  
format, 225  
fsck, 226  
grep, 232  
halsys, 29  
head, 89  
if, 323  
kill, 203  
lc, 92  
lrm, 220  
ln, 105  
logname, 41  
logout, 28  
lpr, 220  
ls, 95  
mail, 268  
mesg, 45  
mkdir, 109  
mkts, 225  
more, 84  
mount, 227  
mv, 108  
news, 40  
nice, 205  
passwd, 51  
pr, 221  
ps, 204  
pwd, 90  
random, 52  
rmdlr, 110  
sdiff, 253  
setkey, 50  
shift, 307  
shutdown, 29  
sort, 241  
sysadmin, 288  
tail, 86  
tar, 290  
tee, 212  
umount, 228  
uname, 42  
uniq, 248  
units, 62  
until, 329  
v1, 125  
wc, 113  
while, 328  
who, 24, 42  
write, 43
QUICK INDEX OF vi COMMANDS

' (quote), 149
" (two quotes), 148
' (back-quote), 149
" (two back-quotes), 148
" (double-quote), 164, 165
" (tilde), 160
, (comma), 144
. (period), 154, 165
: (colon), 128
; (semicolon), 144
? (question mark), 138
! (exclamation mark), 175
!! (two exclamation marks), 175
ls., 175
% (percent), 197
+ (plus sign), 145
- (minus sign), 145
/ (vertical bar), 144
\ (backslash), 138
{ (left brace bracket), 143
} (right brace bracket), 143
( (left parenthesis), 143
) (right parenthesis), 143
< (less-than sign), 138
<< (two less-than signs), 196
> (greater-than sign), 195
>> (two greater-than signs), 196
[][] (two square brackets), 197
]] (two right square brackets), 197
<BACKSPACE>, 146, 157
<CURSOR-DOWN>, 145
<CURSOR-LEFT>, 146
<CURSOR-RIGHT>, 146
<CURSOR-UP>, 145
<ENTER>, 145
<ESC>, 156
<SPACE>, 146
<TAB>, 127
A, 156
a, 155
:ab, 192
:args, 132
B, 143
b, 143
C, 160
c, 159
cc, 160
c:co, 182
:co, 182
D, 164
d, 163
dd, 164
e, 143
e, 143
e:el, 133
 Eli, 133
F, 144
f, 144
G, 137
g, 137
:gl, 183
H, 143
h, 146
i, 156
I, 155
J, 162
j, 146
k, 146
L, 143
l, 146
M, 143
n, 139
n, 133
:nl, 133
O, 156
o, 156
P, 164, 167-168
p, 164, 167-168
Q, 129
q, 131
:q!, 131
R, 159
r, 158
re, 181
rel, 176, 181
rew, 132
:rewl, 132
S, 159
s, 159
*:s, 183
:set, 172
:set al, 194
:set all, 172
:set autoindent, 194
:set autowrite, 173
:set aw, 173
:set ic, 173
:set ignorecase, 173
:set list, 173
:set mesg, 173
:set nu, 174
:set number, 174
:set report, 174
:set shiftwidth, 196
:set sw, 196
:set tabstop, 196
:set term, 174
:set ts, 196
:set wa, 175
:set warn, 174
:set window, 174
:set wm, 160, 161, 174
:set wrapmargin, 174
:set wrapscan, 175
:set writeany, 175
:so, 191
T, 144
t, 144
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>:ta</td>
<td>198</td>
</tr>
<tr>
<td>U</td>
<td>154</td>
</tr>
<tr>
<td>u</td>
<td>154</td>
</tr>
<tr>
<td>:una</td>
<td>192</td>
</tr>
<tr>
<td>vi</td>
<td>125, 198</td>
</tr>
<tr>
<td>:vi</td>
<td>129</td>
</tr>
<tr>
<td>W</td>
<td>143</td>
</tr>
<tr>
<td>w</td>
<td>143</td>
</tr>
<tr>
<td>:w</td>
<td>180</td>
</tr>
<tr>
<td>:wl</td>
<td>131</td>
</tr>
<tr>
<td>:w&gt;&gt;</td>
<td>181</td>
</tr>
<tr>
<td>X</td>
<td>163</td>
</tr>
<tr>
<td>x</td>
<td>163</td>
</tr>
<tr>
<td>:x</td>
<td>130</td>
</tr>
<tr>
<td>Y</td>
<td>165</td>
</tr>
<tr>
<td>y</td>
<td>165</td>
</tr>
<tr>
<td>yy</td>
<td>165</td>
</tr>
<tr>
<td>z</td>
<td>147</td>
</tr>
<tr>
<td>ZZ</td>
<td>136</td>
</tr>
</tbody>
</table>
GENERAL INDEX

aborting a command, 27
aborting a process, 203
argument, 34, 36
ASCII code, 72, Appendix A
background processes, 202
back-quote, 314
backslash, quoting with, 119
BACKSPACE>, 27
backups, 286-294
banner, creating a, 48
bc, 56-61
    advanced calculations, 61
    sample session, 58
    specifying calculations, 57
    using variables, 59-60
bits, 71
Bourne shell, 24
buffer, 16
bytes, 71
C-Shell, 24
calculator program, 56-61
calendar, displaying, 46
checking a file system, 226
command substitution, 314
comparing text files, 248-256
concatenating files, 214
console, reading and writing to, 217
converting money, 65
converting units of measurement, 62-66
copying files, 102-105
correcting as you type, 27
counting lines, words, and characters, 113
CTRL-d>, 27, 28
CTRL-q>, 27
CTRL-s>, 27
CTRL-u>, 27
CTRL> key, 13
cursor, 17
data, 5
date, displaying, 45
DEL>, 27
deamons, 202
display, pausing, 28
directories, 70, 75, 89-100
    changing groups, 265
    changing owners, 265
changing permissions, 263
changing working directory, 90
displaying information about, 91-97
erasing, 110
home directory, 335
making new, 109
permissions, 262
removing, 110
sample session of commands, 97-100
working, 81
working directory, 90
diskettes:
    formatting, 225
    using, 223-230
    using with DOS files, 294-297
dismounting a file system, 223, 228
display, 5
display, restarting, 28
displaying a file, 84-89
DOS, using files, 294-297
echoing, 16, 47
eof, 27, 28
erase, 27
erasing directories, 110
erasing files, 107
.exrc file, 193
factors, calculating prime, 53
file names, 74
file systems, 223
    checking, 226
    dismounting, 223, 228
    making, 225
    mounting, 223, 227
files, 70-133
    binary, 72
    changing groups, 265
    changing owners, 165
    changing permissions, 263
    concatenating, 214
    copying, 102-105
    counting lines, words, and characters, 113
    creating, 74
    definition of, 70
    determining type of information contained by, 113
directories, 70
displaying, 84–89
  displaying beginning of, 89
  displaying end of, 89
  displaying information about, 91–97
erasing, 107
linking, 105–107
moving, 108
names, 74
ordinary, 70
owning, 76
parent directories, 77
pathnames, 80
paths, 80
permissions, 261
removing, 107
renaming, 108
special, 70
sub-directories, 77–79
text, 72
using DOS files, 294
using regular expressions to specify names, 114–119
  with a particular name, searching for, 111
working directory, 81
filters, 210
foreground processes, 202
formatting a diskette, 225
function keys, setting, 50
groups, 260, 265
home directory, setting, 335
input/output, 5
intr. 27
inumber, 75
keyboard, special signals, 27
keyboard buffer, 16
keyboard of IBM PC AT, 11–17
kill. 27
killing a process, 203
link, 75
linking files, 105–107
logging in, 21, 23
  as superuser, 23
  from a remote terminal, 23
logging out, 21, 28
mail. 268–284
  disposing of messages, 274–276
  parts of a message, 269
  receiving messages, 270–274
  sending messages, 276–281
  summary of commands, 282–284
making a file system, 225
making new directories, 109
more, 84–89
  advanced commands, 87
  searching for patterns, 86
  summary of commands, 88
mounting a file system, 223, 227
moving files, 108
newline. 49
news, system, 40
operating system, 3
option, command, 34–35
ordinary files, 70
owner, 265
parent directories, 77
password, 4, 23
  changing your, 50
pathnames, 80
paths, 80
  setting the search path, 336
pausing the screen display, 28
permissions, 260–264
  changing, 263
pipes, 210
"print", use of this term with Xenix, 6
printing, 218–223
priorities, 205
processes, 201
  displaying information, 204
  specifying priorities, 205
  stopping, 203
processid, 201
profile file, 334
program, 5
prompt, 24–26
quoting, 48
  backslash, 119
  summary, 315
random number, generating a, 52
redirecting standard input or output, 207
regular expressions, 114–121
  with grep, 235
  with vi, 140
removing directories, 110
removing files, 107
renaming files, 108
restarting the screen display, 28
root userid, 9
<RUBOUT>, 27
scientific notation, 62
scripts, 208
  shell, 300-338
searching for a file with a particular name, 111
searching text files, 232-239
search path, setting the, 336
sending messages to another userid, 43-45
setting the function keys, 50
setting the search path, 336
setting your terminal type, 336
shell, 24
  choosing a, 26
  comparison of Bourne shell, C-Shell, visual shell, 24
shell, programming the, 298-338
  case, 326
  control flow, 322-334
  exit, 333
  for, 331
  if, 323
  performing tests, 320-322
  until, 329
  using variables, 305-314
while, 328
sorting text files, 240-248
space, 49
special files, 70, 216
  diskettes, 224
standard input, 207
standard output, 207
starting Xenix, 21-24
stopping a command, 27
stopping a process, 203
stopping Xenix, 28
strings, 48-50
sub-directories, 77-79
superuser, 9, 23
syntax, 36-38
system maintenance, 22
system manager, 9
tab, 49
tees, 212
terminal, 7-8
  reading and writing to, 216
  setting your terminal type, 336
time, displaying, 45
tree-structured file systems, 79-81
tty, 8
units program, 62-66
  converting money, 65
  important constants, 65
user, 4
userid, 4, 8
  displaying, 41, 42-43
  sending messages to another, 43-45
vi editor, 122-198
  abbreviations, 192
  absolute move commands, 148
  accessing the shell, 175-177
  automatic indentation, 194-197
  breaking lines, 161
  changing, 158-160
  changing screen position of current line, 147
  changing the case of letters, 160
  command line, 126
  command mode, 128
  copying, 182
  current file, 125
  current line, 135
  delete buffers, 164
  deleting, 163-165, 182
  editing buffer, 126
  editing more than one file, 131-133
  editing programs, 194-198
ex commands, 178-194
executing Xenix commands, 175
.exrc file, 193
how vi uses the screen, 126
insert mode, 128
inserting, 155-158
joining lines, 161
learning, 124
moving, 182
moving the cursor, 135-137, 143-146
named buffers, 165
options, 170-175
regular expressions, 140
repeating commands, 150, 154
replacing, 158-160
returning and marking, 148-150
saving your work, 130
searching for a pattern, 138-142
setting up the work environment, 170-175
setting up your working environment, 193
specifying your terminal type, 170
starting, 125
stopping, 130
substituting, 183-191
summary of commands, Appendix B
tags, 197
undoing commands, 154
unnamed buffer, 167
using an editing script, 191
using ex commands, 128

visual shell, 24

working directory, 81
    changing, 90
    displaying name of, 90

Xenix:
    displaying information about, 42
    editors that come with, 123
    executing commands from vi, 175
    form of commands, 33
    grouping commands, 316-318

how to learn, 32-33
installing, 21
introduction to, 3-9
logging in, 21, 23
logging in as superuser, 23
logging in from a remote terminal, 23
logging out, 21, 28
reference manuals, 31
setting up your working environment, 334-338
starting, 21-24
stopping, 28
syntax of commands, 36-38
MORE COMPUTER BOOKS FROM SCOTT, FORESMAN AND COMPANY

Using Your IBM PC AT
This valuable reference guide introduces you to IBM's hottest personal computer, the PC AT. Harley Hahn clearly shows you how to use DOS 3.3 and 3.1, manage disks and files, use the line editor, and more. $19.95, 272 pages

Working with Xenix System V
Learn to use your Xenix-based computer more effectively. The first guide to the new Xenix System V, this nontechnical book offers a helpful overview of Microsoft Xenix and the shell, and includes a quick-reference command chart. By Moore. $19.95, 256 pages

Short, Simple, and BASIC: Business Programs on the IBM PC for the Smart Manager
Here are 59 useful programs for business and financial analysis on your PC—including programs for breakeven and cash flow analysis, amortization, and more. By Robinson & Kepner. $18.95, 352 pages

The ABCs of Lotus 1-2-3
Put Lotus 1-2-3 to work immediately in your business. Covers the basics, advanced features, and tips not found in the software manual. By Kling. $18.95, 432 pages

To order,
contact your local bookstore or computer store, or send the order form to

Scott, Foresman and Company
Professional Publishing Group
1900 East Lake Avenue
Glenview, IL 60025

In Canada, contact
Macmillan of Canada
164 Commander Blvd.
Agincourt, Ontario
M1S 3C7
Order Form

Send me:

______ Using Your IBM PC, $19.95, 18262
______ Working with Xenix System V, $19.95, 18080
______ Short, Simple, and BASIC, $18.95, 18286
______ ABCs of Lotus 1-2-3, $18.95, 15996

☐ Check here for a free catalog

Please check method of payment:
☐ Check/Money Order  ☐ MasterCard  ☐ Visa

Amount Enclosed $ ___________________________

Credit Card No. _______________________________________

Expiration Date ________________________

Signature _______________________________________

Name (please print) _________________________________

Address __________________________________________

City ________________________ State __________ Zip __________

Add applicable sales tax, plus 6% of the Total for U.P.S.

Full payment must accompany your order. Offer good in U.S. only.

A18260
The only book specifically devoted to Xenix on the IBM PC AT

"An excellent introduction to Xenix on the PC AT, and to Unix in general."—Bart Miller, Assistant Professor, Computer Sciences, University of Wisconsin-Madison

"Corporate Xenix users will undoubtedly need a clear, easy-to-read tutorial on using the operating system, and this book meets those needs very well.... A very non-technical approach to putting Xenix to practical use."—Alan Simpson, author and software consultant

**Mastering Xenix on the IBM PC AT** gives you a thorough introduction to IBM Personal Computer Xenix, one of the most powerful operating systems for small computers.

This definitive book focuses on the practical uses of Xenix and makes a complex system easy to understand. The second book in our popular series on the PC AT, this tutorial is carefully designed for new Xenix users. Harley Hahn provides a clear overview of the operating system, detailed explanations of important concepts, and a wealth of examples.

**Mastering Xenix on the IBM PC AT** starts with the basics and progresses to more advanced Xenix concepts, helping you to
- understand and use the Xenix file system
- work with the major Xenix commands and options
- use the vi editor to create and modify your own text files
- send and receive messages with the Xenix mail system
- program the Xenix shell and more.

Well organized for quick reference, this guide also includes a detailed glossary and a summary of vi commands.

Harley Hahn is a professional writer and author of Using Your IBM PC AT, recently published by Scott, Foresman. A resident of southern California, Mr. Hahn holds an Honors Bachelor of Mathematics degree from the University of Waterloo in Canada, and a Master's degree in Computer Science from the University of California at San Diego. The next book in his PC AT series is on assembly language programming.

Scott, Foresman and Company