Operating Systems and Languages Library

XENIX V

User Guide

olivetti
PERSONAL
COMPUTER

olivetti

PREFACE

This manual is a user guide for the XENIX V operating system. It is for anyone who wishes to use this operating system on an Olivetti Personal Computer.

SUMMARY

The first four chapters of the XENIX V User Guide are devoted to a broad introduction to XENIX's capabilities. There follow chapters on the mail and calculator utilities. The remainder of the guide is devoted to chapters describing the various command interpreters (or "shells") and editors available with the XENIX V system.

RELATED PUBLICATIONS:

M28 Installation and Operations Guide (Code 4024950 J)

XENIX V Installation and System Administration Guide (Code 4022960 S)

XENIX V User and System Administrator Reference Manual (Code 4022320 Z)

XENIX V System and Application Software Development Tools User Guide (Code 4022990 B)

XENIX V System and Application Software Development Tools Reference Manual (Code 4031710 V)

XENIX V Text Processing User Guide (Code 4022980 C)

XENIX V Text Processing Reference Manual (Code 4031720 W)

C-Language under XENIX V User Guide (Code 4022970 T)

C-Language under XENIX V Reference Manual (Code 4033810 Y)

MS-MACRO ASSEMBLER under XENIX V User Guide (Code 4022950 Z)

MS-MACRO ASSEMBLER under XENIX V Reference Manual (Code 4033800 B)

DISTRIBUTION: General (G)

FIRST EDITION: August 1986

PUBLICATION ISSUED BY:

Ing. C. Olivetti & C. S.p.A.
Direzione Documentazione
77, Via Jervis - 10015 IVREA (Italy)

TRADEMARK NOTICE

. OLIVETTI is a trademark of Ing. C. Olivetti & C., S.p.A.
. OLITERM is a trademark of Ing. C. Olivetti & C., S.p.A.
. SORTP is a trademark of Ing. C. Olivetti & C., S.p.A.
. ASM-86 is a trademark of Digital Research
. CB-86 is a trademark of Digital Research
. CLEO is a trademark of Phone 1 Inc.
. ETHERNET is a trademark of Xeros Corp.
. GW is a trademark of Microsoft Corp.
. IBM is registered trademark of International Business Machines Corp.
. MICROSOFT is a registered trademark of Microsoft Corp.
. MS is a trademark of Microsoft Corp.
. OMNINET is a trademark of Corvus System Inc.
. p-System is a trademark of Softech Microsystems, Inc.
. PC-DOS is a trademark of International Business Machines Corp.
. PEACHPACK is a trademark of Peachtree Software International Ltd.
. SID-86 is a trademark of Digital Research
. TerminALL is a trademark of TOPIC System, Inc.
. TOPIC is a trademark of TOPIC System, Inc.
. UNIX is a trademark of Bell Laboratories
. XENIX V is a trademark of Microsoft Corporation
. Z80 is a registered trademark of Zilog Inc.
. Z8000 is a registered trademark of Zilog Inc.

CONTENTS

1. INTRODUCTION

ABOUT THIS CHAPTER

This chapter serves as an introduction to the remainder of the XENIX V User Guide.

CONTENTS

OVERVIEW

This manual introduces the XENIX V System by explaining the fundamental concepts and software needed to use it effectively. The XENIX system is an improved and enhanced version of the UNIX operating system from Bell Laboratories. It is intended for use in schools, corporations, laboratories and small office environments. The system is well known as a productive environment for software development and has been used for many years as a text processing environment.

THE XENIX V SYSTEM

The XENIX V system consists of a general-purpose multiuser operating system and over one hundred utilities and application programs. In addition to the XENIX Operating System described in this manual, two other XENIX V system packages are available: the XENIX V Software Development System and the XENIX V Text Processing System.

THE XENIX WORKING ENVIRONMENT

The XENIX system is built around the XENIX operating system, which organizes and controls computer resources such as memory, disks, lineprinters, terminals, and any other peripheral devices connected to the system. XENIX is a multiuser and multitasking operating system: a multiuser system permits several users to use a computer simultaneously, thus providing lower cost in computing power per user; a multitasking system permits several programs to run at the same time, thereby increasing productivity.

Because UNIX (and thus XENIX) has been accepted as a standard for high-end operating systems, a great deal of software is available for this environment. In addition, XENIX is a bridge to the MS-DOS operating system, the most widely used 16-bit operating system in the world. For systems that support MS-DOS, XENIX provides commands that let you access MS-DOS format files and disks.

Other characteristics of the XENIX system include:

- A powerful command language for programming XENIX commands. Unlike other interactive command languages, the XENIX shell is a full programming language.

- Simple and consistent naming conventions. Names can be used absolutely, or relative to any directory in the file system.

- Device-independent input and output: each physical device, from interactive terminals to main memory, is treated like a file, allowing uniform file and device input and output.

- A set of related text editors, including a full screen editor.

- Flexible text processing facilities. XENIX commands let you find and extract patterns of text from files, compare and find differences

between files, and search through and compare directories. Text formatting, typesetting, and spelling error-detection facilities, as well as a facility for formatting and typesetting complex tables and equations, are also available.

- A sophisticated calculator program.

- Mountable and dismountable file systems that permit addition of diskettes to the file system.

- Flexible directory and file protections that provide the owner of each file or directory (or groups of users) with complete read, write, and execute access.

- Facilities for creating, accessing, moving, and processing files and directories in a simple and uniform way.

The XENIX system also includes enhancements developed at the University of California at Berkeley, and a visual shell suitable for first-time users.

USING THIS MANUAL

SEE	TO
Chapter 1, Introduction	Become acquainted with the XENIX system.
Chapter 2, Demonstration	Gain hands-on experience with the XENIX system.
Chapter 3, Basic Concepts	Become acquainted with fundamental XENIX concepts: the file system, naming conventions, commands, and input and output.
Chapter 4, Tasks	Learn to use XENIX commands to perform everyday tasks.
Chapter 5, Using mail	Learn to use the XENIX mail facility.
Chapter 6, bc: A Calculator	Learn to use **bc** , a sophisticated calculator program.
Chapter 7, The Bourne Shell	Become acquainted with the shell command interpreter and learn to write procedures.

SEE	TO
Chapter 8, The Visual Shell	Become acquainted with the Visual Shell.
Chapter 9, The C-Shell	Become acquainted with the C-Shell.
Chapter 10, Using vi	Learn to use the screen editor, vi .
Chapter 11, Using ed	Learn to use the line editor, ed.
Chapter 12, Editing With sed And awk	Learn to use the editors sed and awk .

2. DEMONSTRATION

ABOUT THIS CHAPTER

This chapter lets you get started with your XENIX V operating system by providing a demonstration of some of its basic features.

CONTENTS

INTRODUCTION

The demonstration in this chapter presents the basic concepts of the XENIX system. You will learn how to:

- log in
- type at your keyboard
- correct typing errors
- enter commands
- log out

BEFORE YOU USE XENIX

Before you can use the system, your name must appear on the XENIX user list. See your system administrator, or refer to the "XENIX Installation and System Administration Guide" for detailed information on adding users to the system.

Once your name is on the XENIX user list, you have a XENIX account. With this account you'll be given a user name, a password, and a login directory. Once you have these, all you need is a terminal from which you can enter, or log into the system. XENIX supports most terminals; see the "XENIX Installation and System Administration Guide" for information on how to configure your terminal.

LOGGING IN

The **login:** prompt should appear on the terminal screen.

To login:

1. Type your login name.
2. Press **CR** .

   If a password is required, you will be asked for it.

3. Type the password.
4. Press **CR** .

The password does not appear on the screen as you type; this prevents others from viewing it.

A successful login produces a "prompt character", a single character that indicates the system is ready to accept commands. The prompt is usually a dollar sign ($) or a percent sign (%). You may also receive a login message such as:

```
you have mail
```

indicating that another system user has sent you mail. Refer to Chapter 5 to find out how to read your mail.

Note: If the system displays nonsense characters when you type, your terminal is probably receiving information at the wrong speed. Check your terminal switches with the **stty** command. (See **stty (C)** in the "XENIX User and System Administrator Reference Manual" for information on setting terminal switches.) If the switches are set correctly, but the problem still occurs, press the **INTERRUPT** key a few times.

TYPING COMMANDS

Once the prompt character appears, the system is ready to respond to commands typed at the terminal. Try typing:

```
date
```

followed by **CR** . The system responds with a line like this:

```
Mon Jun 16 14:17:10 EST 1983
```

Remember to press the **CR** key at the end of each command line. Different terminals have different key-top inscriptions to indicate the **CR** key, such as **ENTER** or **RETURN** , but in all cases the key performs the same function.

Another command you might try is **who**, which lists the names of everyone that is logged in to XENIX. A typical display from the **who** command might look something like this:

```
you         console     Jan 16   14:00
joe         tty01       Jan 16   09:11
ann         tty02       Jan 16   09:33
```

The time, given in the fourth column, indicates when the user logged in; tty *nn* is the system name for each user's terminal, where *nn* is a unique two-digit number. Console is the special name of the system master terminal.

If you make a mistake typing the command name, you will see a message on your screen. For example, if you type:

```
whom
```

the system responds with the message:

```
whom:  not found
```

Note that case is significant in XENIX. The commands:

```
who
```

and:

```
WHO
```

are not the same.

Now try displaying a message on your screen using the **echo** command. Type:

```
echo hello world
```

The **echo** command echoes the rest of the command line to your terminal:

```
hello world
```

Now try this:

```
echo hello world >greeting.file
```

This time the **echo** command sends its output to a new file named greeting.file instead of to your terminal. Note the use of the greater-than sign (>) to redirect the output of the command from your screen to a file.

Now type:

```
ls
```

to list just the name of the file, greeting.file. To look at the contents of greeting.file, type:

```
cat greeting.file
```

cat stands for concatenate. In this instance, the **cat** command sends the following output to your terminal screen:

```
hello world
```

To remove greeting.file, type:

```
rm greeting.file
```

XENIX command names are often shortened to mnemonic names. For example, **ls** is short for list, **cat** is short for concatenate, and **rm** is short for remove.

CORRECTING TYPING ERRORS

Provided you have not pressed **CR** , there are two ways to correct typing errors on a command line:

- Press the **BKSP** to erase the last character typed.

  You can use the **BKSP** key to erase characters back to the beginning of the line, but not beyond.

- Press **CTRL U** to erase all characters on the current input line.

READ-AHEAD AND TYPE-AHEAD

XENIX has full read-ahead, which means that XENIX remembers what you have typed, no matter how quickly or when you type. If you enter characters while a command is displaying text on the screen, XENIX stores and interprets the characters in the correct order, even though they appear to be intermixed with the screen text. Full read-ahead lets you type several commands (i.e., type ahead) one after another without waiting for the first to finish execution. Note that this doesn't work when you log in; type-ahead works only after you have entered your password and received the prompt.

STRANGE TERMINAL BEHAVIOUR

Occasionally, your terminal may act strangely. To fix the problem, try resetting the terminal with the following actions:

- Turn your terminal off, then quickly back on.

- Log out and log back in.

If these procedures don't solve the problem, read the description of the stty (C) command in the "XENIX User and System Administrator Reference Manual" for more information about setting terminal characteristics.

STOPPING A PROGRAM

To stop the execution of most programs and commands, press the **INTERRUPT** key. Inside some programs, like most text editors, typing **INTERRUPT** stops whatever the program is doing without aborting the program itself. Throughout this manual, the phrase "send an interrupt" means to press the **INTERRUPT** key.

LOGGING OUT

To end a XENIX session, you must log out:

1. Move to a new line.

2. Press **CTRL D**

It is not sufficient just to turn off the terminal, since this does not log you out. **CTRL D** also ends some programs, so beware.

LEARNING MORE

The skills you have learned in this demonstration will get you started using XENIX. You will probably want to learn how to use a XENIX shell and a text editor. You use the shell language to speak to the system and execute commands. Text editors let you create and edit files, documents, and programs. Editors and shells are discussed thoroughly later on in this guide.

The next two chapters introduce you to basic XENIX concepts and commands. Be sure to read these chapters; it is especially important to understand how the file system is organized and how files are named.

3. BASIC CONCEPTS

ABOUT THIS CHAPTER

This chapter supplies information about basic XENIX V concepts such as files, directories and commands.

CONTENTS

INTRODUCTION

This chapter introduces you to the basic concepts you need to function in the XENIX environment. After reading this chapter you should understand how the system's files, directories, and devices are organized and named, how commands are entered, and how a command's input and output can be manipulated.

FILES

The file is the fundamental unit of the XENIX file system. In XENIX there are really three different types of files:

- Ordinary files (what we usually mean when we say "file")
- Directories
- Special files

ORDINARY FILES

An ordinary file is simply a named concatenation of 8-bit bytes. Whether these bytes are interpreted as text characters, binary instructions, or program statements is up to the programs that examine them. Ordinary files typically contain textual information such as documents, data, or program sources. Executable binary files are also of this type.

Every ordinary file has the following attributes:

- A filename (not necessarily unique)
- A unique system number (inode number)
- A size in bytes
- A time of creation
- A time of modification
- A time of last access
- A set of access permissions

Access permissions insure file privacy and security. The owner provides read-write-execute permissions to files to control access by the owner, by a group of users, and by anyone else. By default, the owner of a file is its creator. The owner can read the file or write to it. By default, other users can read a file owned by another, but not write to it. File permissions can be altered with the **chmod** command, described in "Changing Permissions" in Chapter 4.

DIRECTORY FILES

Directory files are read-only files containing the name and inode number of each file or directory residing within the given directory. An inode number is a unique number associated with a given file. All files on the system have inode numbers. A name/inode number pair is called a link. You use the **ls** command to examine directory files and to list file information. When paired with the inode number, the **ls** command can find additional information about a file.

Like ordinary files, directories can be by assigned appropriate access permissions to insure privacy and security. By default, the owner of a directory can read, create or remove files within that directory. Similarly by default, a user can read files within the directory of another, but not add or remove them. See "Using File and Directory Permissions" in Chapter 4 for detailed information on access permissions.

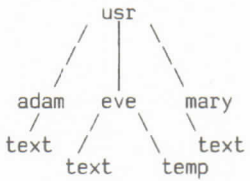
SPECIAL FILES

Special files correspond to physical devices such as hard disks and diskettes, lineprinters, terminals, and system memory. They are called device special files, and are not discussed in this manual.

DIRECTORY STRUCTURE

XENIX organizes all files into a structured directory hierarchy. Each system user has his own personal directory, commonly known as his home directory. A home directory may contain subdirectories over which the user has ownership and control.

A diagram of part a typical user directory is shown below:

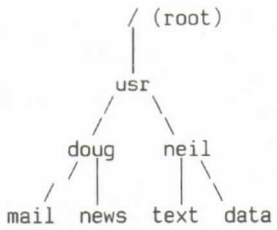


In this example, the usr directory contains each user's personal directory. Notice that Mary's file named text is unrelated to Eve's. When you log in to XENIX, you automatically enter the system at your home directory. Any files you create will initially reside here.

FILE SYSTEMS

A file system is a set of organized files. In XENIX, this set of files consists of all available resources including data files, directories, programs, lineprinters, and disks. Thus, the XENIX file system is a system for accessing all system resources.

The XENIX file system is organized hierarchically. The figure below illustrates a typical file system.



- The root directory, /, is the highest-level directory
- usr, doug, and neil represent directories
- mail, news, text, and data represent data files

NAMING CONVENTIONS

Every file, directory, and device in XENIX has both a filename and an absolute pathname. The pathname is a map of the file or directory's location in the system. The absolute pathname is unique within the entire system; filenames are unique only within directories and need not be unique system-wide.

FILENAMES

A simple filename is a sequence of one to fourteen characters. Every file, directory, and device in the system has a filename. Filenames uniquely identify directory contents. While no two filenames in a directory may be the same, filenames in different directories may be identical.

Although you can use almost any character in a filename, it is best to confine filenames to the alphanumeric characters and the period. The following characters have special XENIX functions and should not be used in filenames:

- Dashes (-)

- Question marks (?)
- Asterisks (*)
- Brackets ([ ])
- All quotation marks (` `` '' ')
- Slashes (/) and backslashes (\)
- Control characters

PATHNAMES

A pathname is a sequence of directory names followed by a simple filename. Elements of the pathname are separated by a slash:

```
/usr/doug/news
```

A pathname beginning with a slash is called a full pathname: it specifies a file that can be found by beginning a search at the root directory. A pathname without an initial slash is called a relative pathname: it specifies files that can be found by beginning the search at the user's current directory (also known as the working directory).

SAMPLE NAMES

Some sample names follow:

/	The absolute pathname of the root directory of the entire file system.
/bin	The directory containing most of the frequently used XENIX commands.
/usr	The directory containing each user's personal directory. The subdirectory, /usr/bin contains frequently used XENIX commands not in /bin.
/dev	The directory containing files corresponding to physical devices (e.g., terminals, lineprinters, and disks).
/dev/console	The name of the system master terminal.
/dev/tty	The name of the user's terminal.
/lib	The directory containing files used by some standard commands.
/tmp	This directory contains temporary scratch files.

/usr/joe/project/A	A typical full pathname; this one happens to be a file named A in the directory named project belonging to the user named joe.
bin/x	A relative pathname; it names the file x in subdirectory bin of the current working directory. If the current directory is /, it names /bin/x. If the current directory is /usr/joe, it names /usr/joe/bin/x.
file1	Name of an ordinary file in the current directory.

In the XENIX system, each user resides in a directory called the current directory. All files and directories have a parent directory. This directory is the one immediately above, which contains the given file or directory. The XENIX file system provides special shorthand notations for this directory and for the current directory:

. The shorthand name of the current directory. Thus ./filexxx names the same file as filexxx, if such a file exists in the current directory.

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

SPECIAL CHARACTERS

XENIX lets you use special characters to specify a sequence of names containing a common pattern:

CHARACTER	MATCHES
*	zero or more characters
[]	any character inside the brackets
?	any single character

For example, to list the names of some similarly named files, you might type:

```
ls chap*
```

This produces:

```
chap1.1
chap1.2
chap1.3
...
```

The asterisk can be used anywhere in the filename, and can occur several times. An asterisk by itself matches all filenames not containing slashes or beginning with periods, so:

```
cat *
```

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

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

```
ls chap[12349]*
```

The brackets ([and]) mean match any of the characters inside the brackets. You can even abbreviate a range of consecutive letters or digits:

```
ls chap[1-49]*
```

(Note that this does not match forty-nine filenames, but only five.)

To indicate a range of letters, enclose them within brackets:

```
[a-z]
```

This matches any character in the range "a" through "z".

The question mark (?) matches any single character, so:

```
ls ?
```

lists all files that have single-character names, and:

```
ls -l chap?.1
```

lists information about the first file of each chapter (i.e., chap1.1 , chap2.1 , ...).

To use the special characters (*, ?, and [...]) literally, enclose the entire argument in single quotation marks. For example, the following command will print out only files named ? rather than all one-character filenames:

```
ls '?'
```

Pattern-matching features are discussed further in Chapter 7, "The Bourne Shell".

COMMANDS

You use commands to invoke executable programs. XENIX reads the command line that you have typed, looks for a program with the given name, and then executes the program if it finds it. Command lines may also contain arguments that specify options or files that the program may need. The next two sections discuss the command line and command syntax.

COMMAND LINE

Whether you are typing commands at a terminal, or XENIX is reading commands from a file, XENIX always reads commands from command lines. The command line is a line of characters that is read by the shell command interpreter to determine what actions to perform. This interpreter, or shell, reads the names of commands from the command line, finds the executable program corresponding to the name of the command, then executes that program. When the program finishes executing, the shell resumes reading the command line. Thus, when you are typing at a terminal, you are editing a line of text called the command-line buffer that becomes a command line only when you press **CR** . You can edit the command-line buffer with the **BKSP** and **CTRL U** keys.

Once you press **CR** , the command-line buffer is submitted to the shell as a command line. The shell reads the command line and executes the appropriate command. If you press **INTERRUPT** before you press **CR** , the command-line buffer is erased.

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

```
date ; pwd
```

To process a command in the background, append an ampersand (&) to the command line. This mode of execution is similar to batch processing on other systems. The main advantage to placing commands in the background is that you can execute other commands from your terminal in the foreground while the background commands execute. Thus:

```
du /usr >diskuse&
```

determines the disk usage in the directory /use, a fairly time-consuming operation, without tying up your terminal. Note that the output is placed in the file diskuse by redirecting output with the greater-than symbol. Redirection is discussed later in this chapter.

SYNTAX

The general syntax for commands is as follows:

command [*switches*][*arguments*][*filenames*]

By convention, command names are lowercase.

Switches, also called options, are optional and usually precede other arguments and filenames. Switches consist of a dash prefix (-) and an identifying letter. For example, the ls command's -r switch:

```
ls -r
```

specifies a directory listing in reverse alphabetical order.

Arguments provide additional information that help the command carry out its task.

Filenames specify the name of files required by the command.

Italics indicate that the words they make up are not to be typed as shown, but represent variable information. The square brackets ([]) indicate that the syntax elements they contain are optional.

INPUT AND OUTPUT

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

```
cat
```

This command now expects input from your keyboard. As input, it accepts as many lines of text as you type until you press **CTRL D** as an end-of-file or end-of-transmission indicator.

For example, type:

```
this is two lines
of input
  CTRL D
```

When you press **CTRL D** , input ends and output begins. The **cat** command immediately outputs the two lines you typed and, since output is sent to the terminal screen by default, that is where the two lines are sent. Thus, the complete session will look like this on your terminal screen:

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

```
$
```

You can redirect command input and output so that input comes from a file instead of from the terminal keyboard and output goes to a file or lineprinter instead of to the terminal screen. You can also create pipes, which allow the output from one command to become the input to another.

REDIRECTION

In XENIX, a file can replace the terminal for either input or output. For example:

```
ls
```

displays a list of files on your terminal screen. But if you say:

```
ls >filelist
```

XENIX places the list of your files in the file called filelist (which is created if it does not exist). If filelist does exist, its contents will be overwritten by this command. The symbol for output redirection, the greater-than sign (>), means put the output from the command into the following file, rather than display it on the terminal screen.

As another example of output redirection, you can combine several files into one by capturing the output of **cat** in a file:

```
cat f1 f2 f3 >temp
```

The output append symbol (>>) operates very much like the output redirection symbol, except that it means add to the end of. So:

```
cat file1 file2 file3 >>temp
```

means concatenate file1, file2, and file3 to the end of whatever is already in temp, instead of overwriting and destroying the existing contents. As with normal output redirection, if "temp" doesn't exist, XENIX creates it.

In a similar way, the input redirection symbol (<) means take the input for a program from the following file, instead of from the terminal. Thus, you could make a script of editing commands and put them into a file called "script". Then you could execute the commands in the script on a file by typing:

```
ed file <script
```

As another example, you could use **ed** to prepare a letter in the file "letter.txt", then send it to several people with:

```
mail adam eve mary joe <letter.txt
```

PIPES

One of the major innovations of the XENIX system is the concept of a pipe. A pipe lets you connect the output of one command to the input of another, so that the two run as a sequence of commands called a pipeline.

For example:

```
sort frank.txt george.txt hank.txt
```

combines the three files named frank.txt, george.txt, and hank.txt, then sorts the output. Suppose that you want to then find all unique words in these files and view the result. You could type:

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

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

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

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

4. TASKS

ABOUT THIS CHAPTER

This chapter shows how to perform common XENIX V tasks such as manipulating files and requesting system information.

CONTENTS

INTRODUCTION

This chapter explains how to perform common tasks on XENIX. The individual commands used to perform these tasks are discussed more thoroughly in the "XENIX user and System Administrator Reference Manual".

GAINING ACCESS TO THE SYSTEM

To use the XENIX system, you must first gain access to it by logging in. When you log in you are placed in your own personal working area. Logging in, changing your password, and logging out are described below.

LOGGING IN

Before you can log in to the system, you must be given a system account. Your name must be added to the user list, and you must be given a password and a mailbox.

Depending on how your system is administered, you may have to add your name to the user list yourself, or someone else may be assigned this task. If you must add your own account to the system, see the "XENIX Installation and System Administration Guide" and **mkuser(C)** in the "XENIX User and System Administrator Reference Manual" for more information. This section assumes your account has already been set up.

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

When the login: prompt appears, follow these steps:

1. Type your login name and press **CR** . If you make a mistake, press **CTRL U** to start the line again. After you press **CR** the message **Password:** appears on your screen.

2. Type your password carefully, then press **CR** . The letters do not appear on your screen as you type, and the cursor does not move. If you make a mistake, press **CR** to restart the login procedure.

If you have typed your login name and password correctly the prompt character appears on the screen. This is usually a dollar sign ($). The prompt tells you that the XENIX system is ready to accept commands from the keyboard.

If you make a mistake, the system displays the message:

```
Login incorrect
login:
```

If you receive this message, repeat the login procedure. You must type all the letters of your user name and password correctly before you are given access to the system; XENIX does not allow you to correct mistakes during the login procedure.

When you log in you may see a message that says something like "Welcome to XENIX", or an announcement that is of interest to all users.

LOGGING OUT

The logout procedure is simple. All you need to do is move the cursor to a line by itself, and press:

CTRL D

In general, **CTRL D** signifies the end-of-file in XENIX, and is often used within programs to signal the end of input from the keyboard. In such cases, **CTRL D** will not log you out; it will simply terminate input to a particular program if you are within that program. This means that it may sometimes be necessary to press **CTRL D** several times before you can log yourself out. For example, if you are in the **mail** program you must press **CTRL D** once to exit the mail program, then again to log out.

CHANGING YOUR PASSWORD

To prevent unauthorized users from gaining access to the system, each authorized user must have a password. When you are first given an account on a XENIX system you are assigned a password by the system administrator. Some XENIX systems require that you change your password at regular intervals. Whether yours does or not, it is a good idea to change your password regularly to maintain system security.

Use the **passwd** command to change your password:

1. Type:

   passwd

   and press **CR** . The following message appears:

   Changing password for user:
   Old password:

2. Carefully type your old password.

   Your password doesn't appear on the screen. If you make a mistake, press **CR** . The message **Sorry** appears, followed by the system prompt. Begin again with Step 1.

   When you have typed your old password, the message:

   New password:

   appears.

3. Type in your new password and press CR .

   The message:

   Retype new password:

   appears.

4. Type your new password again.

   If you make a mistake, press CR . The message:

   Mismatch -- password unchanged

   appears, and you must begin again with Step 1. When you have completed the procedure, the system prompt appears.

CONFIGURING YOUR TERMINAL

On most systems, the standard console terminal is already configured for use with XENIX. However, other terminals of various types may be connected to a XENIX system. In these cases it is important to know how to set terminal options and how to specify the terminal you are using. You may also want to change the configuration of the standard console terminal. The following section discusses these topics.

CHANGING TERMINALS

If you log in to XENIX on a terminal of a type different than the terminal you normally use, you may need to change the shell **TERM** variable. This is normally set to the proper terminal when you log in, but if you switch terminal types you must reset the **TERM** variable. To reset this variable, type the following line at command level:

TERM= *termname*

where *termname* is the name of a known terminal. A list of known terminals is described in **terminals (M)** in the "XENIX User and System Administrator Reference Manual". A variety of terminals are supported; terminal capabilities are listed in the system file /etc/termcap.

SETTING TERMINAL OPTIONS

You can set a number of terminal options with the **stty** command. When entered without parameters, **stty** displays the current terminal settings. For example, typical output might look like this:

```
speed 9600 baud
erase "^h" ; kill "^u"
even -nl
```

Each of the characteristics can be set with **stty**. For more information,

see **stty** (C) in the "XENIX User and System Administrator Reference Manual".

EDITING THE COMMAND LINE

When you sit in front of a terminal and type commands at your keyboard, there are a number of special keys that you can use.

ENTERING A COMMAND LINE

To enter a command line, type the characters, then press **CR** . Once you have pressed **CR** , the computer first reads the command line and then executes it.

ERASING A COMMAND LINE

To erase the current command line, press **CTRL U** .

HALTING SCREEN OUTPUT

Contents of a long file may scroll off the screen faster than you can examine them. To temporarily halt a program's output to the terminal screen, press **CTRL S** . To resume output, press **CTRL Q** .

MANIPULATING FILES

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

CREATING A FILE

To create a file and place text in it, use the editor **vi**, described in Chapter 10. In general, new files are created by commands as needed.

DISPLAYING FILE CONTENTS

The **more** command displays the contents of a file one screenful at a time, and is useful for looking at a file when you don't want to make changes to it. It has the form:

more *options filename*

You can invoke **more** with the following options:

+ *linenumber* Begins the display on the designated line in the file.

+ *text*	Begins the display two lines before *text* , where *text* is a word or number. If *text* is two or more words, they must be enclosed in double quotation marks.
-c	Redraws the screen instead of scrolling.
-r	Displays control characters, which are normally ignored by **more**.

To begin looking at the file "memo" at the first occurrence of the words "net gain", for example, type:

```
more +/"net gain" memo
```

If the file is more than one screenful long, the percentage of the file that remains is displayed on the bottom line of the screen. To look at more of the file, use the following scrolling commands:

CR	Scrolls down one line
d	Scrolls down one-half screen
SPACE	Scrolls down a full screen
n **SPACE**	Scrolls down n lines
.	Repeats the previous command

You cannot scroll backward, toward the beginning of the file.

You can search forward for patterns in **more** with the slash (/) command. For example, to search for the pattern "net gain", type:

```
/net gain/
```

and press **CR** . **more** displays the message:

```
skipping...
```

at the top of the screen, and scrolls to a location two lines above "net gain".

If you are looking at a file with **more** and decide you want to change the file, you can invoke the **vi** editor by typing:

```
v
```

See Chapter 10 for more information on using **vi**.

more quits automatically when it reaches the end of a file. To exit **more** before the end of a file, type:

```
q
```

The **head** and **tail** commands display the first and last ten lines of a

file, respectively. They are useful for checking the contents of a particular file.

For example, to look at the first ten lines of the file memo, type:

```
head memo
```

You can also specify how many lines the head and tail commands display. For example:

```
tail -4 memo
```

displays the last four lines of memo.

The **cat** command also displays the contents of a file. **cat** scrolls the file until you press **CTRL S** to stop it. Pressing **CTRL Q** will continue the scrolling. **cat** stops automatically at the end of a file. To stop the display before the end of the file, press **INTERRUPT** . To display the contents of one file, type:

```
cat file1
```

To display the contents of more than one file, type:

```
cat file1 file2 file3
```

COMBINING FILES

You can use the **cat** command to combine multiple files into one new file. To combine two files named file1 and file2 into a new file named bigfile, type:

```
cat file1 file2 >bigfile
```

The greater than sign (>) redirects output of the **cat** command to the new file. The original files remain intact.

You can also use **cat** to append one file to the end of another. For example, to append file1 to file2, type:

```
cat file1 >>file2
```

file1 still exists as a separate entity.

MOVING A FILE

The **mv** command moves a file into another file in the same directory, or into another directory. For instance, to move a file named text to a new file named book, type:

```
mv text book
```

To move a file into another directory, give the name of the destination directory as the final name in the **mv** command. For instance, to move file1 and file2 into the directory named /tmp, type:

```
mv file1 file2 /tmp
```

The two files no longer exist in your working directory, but now exist in the directory /tmp. The above command has exactly the same effect as typing the following two commands:

```
mv file1 /tmp/file1
mv file2 /tmp/file2
```

If the last argument is the name of a directory, all files designated by filename arguments are moved into that directory.

RENAMING A FILE

To rename a file, simply move it to a file with the new name. The old filename is removed:

```
mv oldname newname
```

COPYING A FILE

There are two forms of the **cp** command: one in which files are copied into a directory, and another in which a file is copied to another file. Thus, to copy three files into a directory named filedir, type:

```
cp file1 file2 file3 filedir
```

In the above command, three files are copied into the directory filedir; the original versions still reside in the working directory. Note that the filenames are identical in the two directories. If the last argument is the name of a directory, all files designated by filename arguments are copied into that directory.

To create two copies of a file in your own working directory, you must rename the copy:

```
cp file filecopy
```

Two files with identical contents reside in the working directory.

To learn how to copy directories, see "Copying a Directory" later in this chapter.

DELETING A FILE

To delete or remove files from your working directory, type:

```
rm file1 file2
```

To remove files interactively, use a command of the form:

```
rm -i file1 file2
```

The system will ask you to confirm your intentions:

- Type "y" and press **CR** to remove the indicated file.
- Type "n" and press **CR** to retain the indicated file.

FINDING FILES

The **find** command searches for files that have a specified name. **find** is useful for locating files that have the same name, or just for finding a file if you have forgotten which directory it is in. The command has the form:

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

pathname is the pathname of the directory you want to search. **find** searches recursively; that is, it starts at the named directory and searches through all files and subdirectories under the directory specified in *pathname* .

The **-name** option indicates that you are searching for files that have a specific filename. For other search options you can use with **find**, see **find(C)** in the XENIX User and System Administrator Reference Manual.

filename is the name of the file you are searching for.

The **-print** option indicates that you want to print the pathnames of all files that match *filename* on your terminal screen. To direct this output to a file instead of your screen, use the output redirection symbol, >.

For example, the following command finds every file named memo in the directory /usr/joe and all its subdirectories:

```
find /usr/joe -name memo -print
```

The output might look like this:

```
/usr/joe/memo
/usr/joe/accounts/memo
/usr/joe/meetings/memo
```

/usr/joe/mail/memo

LINKING FILES

The ln command links two files so that changes to one file are reflected in both. This can be useful if several users need to share information, or if you want a file to appear in more than one directory. This command has the form:

ln *file newfile*

where *file* is the original file, and *newfile* is the new, linked file.

For example, the following command links "memos" in /usr/joe to "joememos" in /usr/mary:

ln /usr/joe/memos /usr/mary/joememos

Whenever /usr/joe/memos is updated, the file /usr/mary/joememos is also changed.

There are three things to remember about linking files:

1. Linking large sets of files to other parallel files can save a considerable amount of disk space.

2. Linking files that are used by more than one person is risky, because any party can alter the file and thus affect the contents of all files linked to it.

3. Removing a file from a directory does not remove other links to the file. Thus the file is not truly deleted from the system. For example, if you delete a file that has 4 links, 3 links remain.

For more information about linking, see ln(C) in the XENIX User and System Administrator Reference Manual.

MANIPULATING DIRECTORIES

Because of the hierarchical organization of the file system, there are many directories and subdirectories in the XENIX system. Within the file system are directories for each user of the system. Within your user directory you can create, delete, and copy directories. Commands that let you manipulate directories are described in the following sections.

PRINTING THE NAME OF YOUR WORKING DIRECTORY

All commands are executed relative to a "working" directory. The name of this directory is given by the **pwd** command, which stands for "print working directory". For instance, if your current working directory is /usr/joe, when you type:

```
pwd
```

you will receive the output:

```
/usr/joe
```

You should always think of yourself as residing "in" your working directory.

LISTING DIRECTORY CONTENTS

You can list the contents of a directory with the **lc** command. This command sorts the names of files and directories in a given directory, and lists them in columns. If no directory name is given, **lc** lists the contents of the current directory. The **lc** command has the form:

lc *option name*

For example, to list the contents of the directory work, type:

```
lc work
```

Your output might look like this:

```
accounts    meetings    notes
mail        memos       todo
```

If no *name* is specified, **lc** lists the contents of the current directory.

The following options control the sort order and the information displayed by the **lc** command:

-a Lists all files in the directory, including the "hidden" files (filenames that begin with a dot, such as .profile and .mailrc).

-r Lists names in reverse alphabetical order.

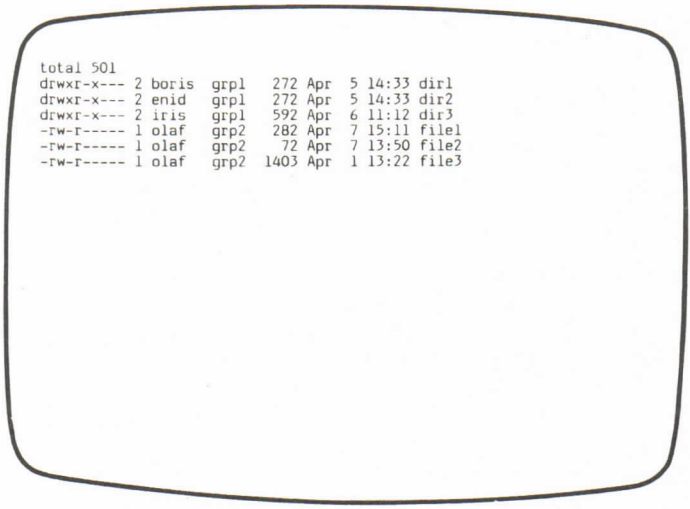
-t Lists names in order of last modification, beginning with the latest (most recently modified). When used with **-r**, lists the oldest first.

-F Marks directories with a backslash (\) and executable files with an asterisk (*).

-R Lists all files and directories in the current directory, plus each file and directory below the current one. The

"R" stands for "recursive".

The -l option displays a "long" directory listing, producing output that might look something like this:



From left to right, the information given for each file or directory includes:

- Permissions
- Number of links
- Owner
- Group
- Size in bytes
- Time of last modification
- Filename

"Using File and Directory Permissions" later in this chapter discusses how to use this additional directory information.

For more information about listing the contents of a directory, see ls(C) in the XENIX User and System Administrator Reference Manual.

CREATING A DIRECTORY

To create a subdirectory in your working directory, use the **mkdir** command. For example, to create a new directory named phonenumbers, type:

```
mkdir phonenumbers
```

REMOVING A DIRECTORY

To remove a directory from your working directory, use the **rmdir** command. For example, to remove the directory named phonenumbers from the current directory, type:

```
rmdir phonenumbers
```

A directory must be empty before it can be removed; this prevents accidental deletions of files and directories.

MOVING A DIRECTORY

Use the **mv** command to move directories. This command has the form:

```
mv oldirectory newdirectory
```

where newdirectory is a directory that already exists. For example, to move the directory /usr/joe/accounts into /usr/joe/overdue type:

```
mv /usr/joe/accounts /usr/joe/overdue
```

The new pathname of /usr/joe/accounts is /usr/joe/overdue/accounts.

RENAMING A DIRECTORY

To rename a directory, use the **mv** command. For example, to rename the directory little.dir to big.dir, type:

```
mv little.dir big.dir
```

This is a simple renaming operation; no files are moved.

COPYING A DIRECTORY

The **copy** command copies directories. This command has the form:

copy *options olddir newdir*

The **copy** command has the following options:

-l Links the copied files to the original.

-m Gives the copied files the same modification dates as the original files.

-r Copies the directory recursively, i.e., copies all the directories under the named directory.

To copy all the files in a directory called /usr/joe/memos into one called /usr/joe/notes, type:

```
copy /usr/joe/memos /usr/joe/notes
```

MOVING IN THE FILE SYSTEM

Each directory in the XENIX system should be thought of as a place that you can move into or out of. This place is called either your working directory or current directory. The commands used to find out where you are and to move around in the structure are discussed below.

FINDING OUT WHERE YOU ARE

Your current location in the file system is the name of the working directory. You can find out this name by using the **pwd** command, which stands for "print working directory". For example, if you are in the directory /usr, then typing the command:

```
pwd
```

prints out the name:

```
/usr
```

CHANGING YOUR WORKING DIRECTORY

Your working directory represents your location in the file system: it is "where you are" in XENIX. To alter this location in the XENIX file system, use the change directory (**cd**) command:

```
cd
```

This changes your working directory to your home directory. To move to any other directory, specify that directory as an argument to **cd**, as in the following command:

```
cd /usr
```

To move up one directory from your current directory, type:

```
cd ..
```

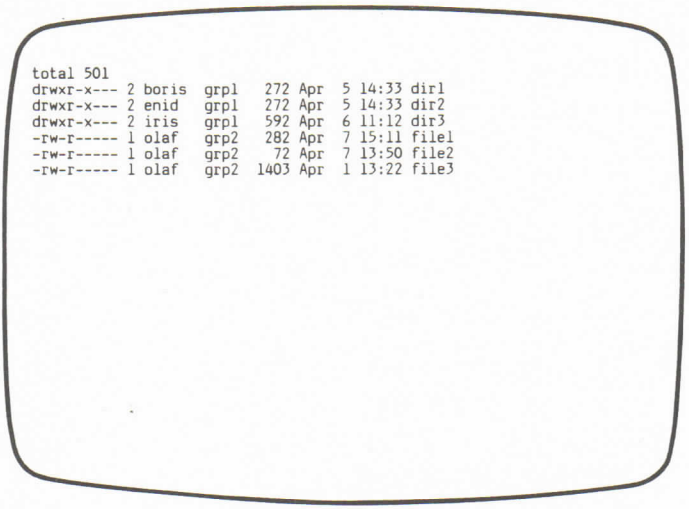
For example, the above command would move you from the directory /usr/joe/work to /usr/joe. Similarly, the command:

```
cd ../..
```

would move you from the directory /usr/joe/work to /usr, moving you up two directories.

USING FILE AND DIRECTORY PERMISSIONS

The XENIX system allows the owner to restrict access to files and directories, limiting who can read, write and execute files under his/her ownership. To determine the permissions associated with a given file or directory, use the ls -l command. The output from the ls -l command should look something like this:



Permissions are indicated by the first ten characters of the output. The permissions for dir1, the first file in the above list, are:

```
drwxr-x---
```

The first character indicates the type of file and must be one of the following:

CHARACTER	TYPE OF FILE
-	ordinary file
d	directory
c	character special device such as a lineprinter of terminal

CHARACTER	TYPE OF FILE
b	block special device such as a hsrd or floppy disk
s	XENIX 3.0 semaphore
m	XENIX 3.0 shared data file
p	named pipe

The next nine characters represent three sets of three permissions:

- Owner permissions
- Group permissions
- All other user permissions

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

Ordinary file permissions have the following meanings:

r The file is readable.

w The file is writeable.

x The file is executable.

- The indicated permission is not granted.

Directory permissions have the following meanings:

r Files can be listed in the directory; the directory must also have "x" permission.

w Files can be created or deleted in the directory; as with "r," the directory itself must also have "x" permission.

x The directory can be searched. A directory must have "x" permission before you can move to it with the **cd** command access a file within it, or list the files in it. Remember that a user must have "x" permission to do anything useful to the directory.

The following are some typical directory permission combinations:

d-------- No access at all. This is the mode that denies access to the directory to a class of users.

drwx------ Allows access by the owner to use **lc**, create files, delete files, access files (subject to file permissions), and **cd**

to the directory. This is the typical permission for the owner of a directory.

drwxr-x--- Allows access by members of the group to use **lc**, and access files subject to file permissions. Group members can **cd** to this directory, but cannot create or delete files in it. This is the typical permission an owner gives to others who need access to files in his directory.

drwx--x--x With these permission settings users other than the owner cannot use **lc** but can **cd** to the directory. Other users can only access a file within this directory by its exact name; they cannot use special characters. Files cannot be created or deleted in the directory by anyone except the owner. This mode is rarely used, but can be useful if you want to give someone access to a specific file in a directory without permitting access to other files in the same directory.

This section discusses ordinary files, executable files, and directories only. For information about other types of files, see **ls(C)** in the XENIX User and System Administrator Reference Manual.

CHANGING PERMISSIONS

The **chmod** command changes the read, write, execute, and search permissions of a file or directory. This command is useful if you have created a file in one mode, but want to give others permission to read, write or execute it. The **chmod** command has the form:

chmod *instruction filename*

The instruction segment of the command indicates which permissions you want to change for which class of users. There are four classes of users:

u User, the owner of the file or directory.

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

o Other, all users of the system.

a All classes of users.

There are three types of permissions:

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

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

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

For example, assume file1 exists with the following permissions:

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

The owner of the file has read and write permission, group members have read permission, and others have no access at all.

To give file1 read permission for all classes of users, type:

```
chmod a+r file1
```

In the instruction segment of the command (a+r) the "a" stands for "all". The resulting permissions are:

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

If file1 had the attributes:

```
-rw-------
```

the following command would give write and execute permissions to members of a group only:

```
chmod g+wx file1
```

The permission attributes would then look like this:

```
-rw--wx---
```

To remove write and execute permission by the user (owner) and group associated with file1, type:

```
chmod ug-wx file1
```

CHANGING DIRECTORY SEARCH PERMISSIONS

Directories also have an execute permission. This attribute signifies search permission, rather than execute permission, since directories cannot be executed. If this permission is denied to a particular user, then that user cannot even list the names of the files in the directory.

For example, assume that the directory dir1 has the following attributes:

```
drwxr-xr-x
```

To prevent users from examining dir1, type:

```
chmod o-xr dir1
```

to remove search permission.

The new attributes for dir1 are:

```
drwxr-x---
```

PROCESSING INFORMATION

In many cases, files will contain information that you may want to process. Various utility programs exist in XENIX to process information. A set of these programs and their uses are described in the following sections.

COMPARING FILES

The **diff** command prints out those lines that differ between two specified files.

For example, suppose a file named men has the contents:

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

and a file named women has the contents:

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

the command:

```
diff men women
```

would produce the following result:

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

You can use the **diff3** command to compare three files. For information about **diff3** see **diff3(C)** in the XENIX User and System Administrator Reference Manual.

ECHOING ARGUMENTS

The **echo** command echos arguments to the standard output device. For example, typing:

```
echo hello
```

outputs:

```
hello
```

on the terminal screen. To output several lines of text, enclose the echoed argument in double quotation marks and press **CR** between lines. A secondary prompt will appear until you type the final double quotation mark. For example, type:

```
echo "Now is the time
For all good men
To come to the
Aid of their party."
```

This will output:

```
Now is the time
For all good men
To come to the
Aid of their party.
```

echo is particularly useful when you program in the shell command language. For more information about the shell, see Chapter 7, "The Shell".

SORTING A FILE

The **sort** command sorts the lines of a file according to the ASCII collating sequence (i.e., it alphabetizes them).

For example, to sort a file and see the results on screen, type:

```
sort filename
```

You can redirect the results of a sort to a new file:

```
sort phonelist >phonesort
```

sort is useful for sorting the output from other commands. For example, to sort the output from execution of a **who** command, type:

```
who | sort >whosort
```

This command takes the output from **who** , sorts it, and then sends the sorted output to the file whosort.

A wide variety of options are available for **sort.** For more information, see **sort(C)** in the XENIX User and System Administrator Reference Manual.

SEARCHING FOR A PATTERN IN A FILE

The **grep** command selects and extracts lines from a file, printing only those lines that match a given pattern.

For example, to print out all lines containing the word "tty38", type:

```
grep "ty38" filename
```

You should always enclose the pattern you are searching for in single quotation marks (') , so that special metacharacters are not expanded unexpectedly by the shell.

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

```
grep "joe" phonelist | grep "822-" >joes.number
```

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

For more information about **grep**, its relatives **fgrep** and **egrep**, and the types of patterns it can be used to search for, see **grep(C)** in the XENIX User and System Administrator Reference Manual.

COUNTING WORDS, LINES, AND CHARACTERS

wc is a utility for counting words in a file. The letters "wc" stand for word count. Words are presumed to be separated by punctuation, spaces, tabs, or newlines. **wc** also counts characters and lines; all three counts are reported by default. For example, to count the number of lines, words, and characters in the file textfile, type:

```
wc textfile
```

Typical output describing lines, words and characters might be:

```
4432    18188   97808   textfile
```

To specify a count of characters, words, or lines only, you must use an appropriate mnemonic switch. To illustrate, examine the following three commands and the output produced by each:

```
wc -c textfile
        97808 textfile

wc -w textfile
        18188 textfile

wc -l textfile
        4432 textfile
```

The first example prints out the number of characters in textfile, the second prints out the number of words, and the third prints out the number of lines.

DELAYING A PROCESS

The **at** program lets you set up commands to be executed at a specified time. It is useful if you want to execute a command when you are not planning to be at your terminal, or even logged in.

The **at** command has the form:

at *time day file*

time is the time of day, in digits, followed by "am" or "pm". One- and two-digit numbers are interpreted as hours, three- and four-digit numbers as hours and minutes. More than four digits are not permitted.

day is optional. It is either a month name followed by a day number, or a day of the week. If no *day* is specified, the command will be executed today.

file is the name of the file that contains the command or commands to be executed.

For example, to find out what processes are running at 10 pm on Tuesday, place the following line in a file named use:

```
ps -a > /usr/myname/use
```

(See Chapter 10, "Using vi", for information on creating and inserting text into files.)

After you have written out the file and returned to command level, type:

```
at 10pm tues use
```

Press **CR** . The XENIX prompt reappears and you may continue working. At 10 pm on Tuesday, XENIX will execute **ps** and place the output in the file use. **at** is unaffected by logging out.

To check what files you have waiting to be processed, use the **atq** command. **atq** lists the files to be processed, along with the following information:

- The file 's user ID
- The file 's ID number
- The date and time the file will be processed

To cancel an **at** command, first check the list of files to be processed and note the file ID number. Then use the **atrm** command to remove the file or files from the list. The **atrm** command has the form:

atrm *number*

For example:

```
atrm 84.032.2300.21
```

removes file number 84.032.2300.21, canceling whatever commands were included in that file. A user can remove only his own files.

CONTROLLING PROCESSES

In XENIX, several processes can run at the same time. For example, you may run the **sort** program on a file in the background, and edit another file in the foreground while the **sort** program is running. Processes you directly control from your keyboard are called foreground processes. Other processes, which you can initiate but otherwise have little control over, are called background processes. At any one time you can have only one foreground process executing, but multiple background processes may execute simultaneously. This section describes procedures for controlling foreground and background processes.

PLACING A PROCESS IN THE BACKGROUND

Normally, commands sent from the keyboard are executed in strict sequence; one command must finish executing before the next can begin. A background process, in contrast, need not finish executing before you give your next command. Background processing is especially useful for commands that may take a long time to complete.

To place a process in the background, type an ampersand (&) at the end of the command. For example, to count the number of words in several large files while simultaneously continuing with whatever else you have to do, type:

```
wc file1 file2 file3 >count&
```

Output is collected in the file count. If output were not put in count, it would appear on the screen at unpredictable times as you continue with your work.

When processes are placed in the background, you lose control of them as they execute. For instance, pressing **INTERRUPT** does not stop a background process. You must use the **kill** command to terminate a background process.

KILLING A PROCESS

To stop execution of a foreground process, press your terminal's **INTERRUPT** key. This kills whatever foreground command is currently running. To kill all processes executing in the background, type:

```
kill
```

To kill only a specified process executing in the background, first type:

```
ps
```

ps displays the Process Identification Numbers (PIDs) of your existing processes, for example:

```
PID   TTY   TIME    CMD
3459   03    0:15    -sh
4831   03    1:52    cc program.s
5185   03    0:00    ps
```

In the above example, you might type:

```
kill 4831
```

where 4831 is the PID of the process that you want killed.

Note

Killing a process associated with the **vi** editor may leave the terminal in a strange mode. Also, temporary files that are normally created when a command starts, and then deleted when the command finishes, may be left behind after a **kill** command. Temporary files are normally kept in the directory /tmp. This directory should be checked periodically and old files deleted.

GETTING STATUS INFORMATION

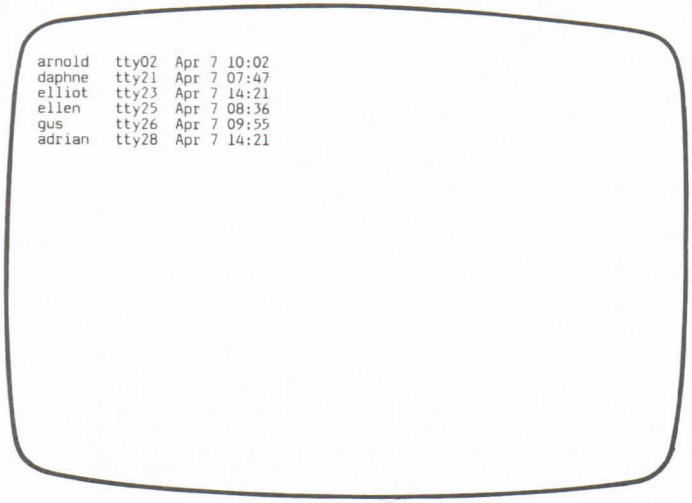
Because XENIX is a large, self-contained computing environment, there are many things that you may want to find out about the system itself, such as who is logged in, how much disk space there is, what processes are currently running. This section explains how to obtain system information.

FINDING OUT WHO IS ON THE SYSTEM

The **who** command lists the names, terminal line numbers, and login times of all users currently logged on to the system. For example, type:

```
who
```

This command produces output such as:



The **finger** command provides more detailed information, such as office numbers and phone extensions. For more information, about using **finger** see **finger(C)** in the XENIX User and System Administrator Reference Manual.

FINDING OUT WHAT PROCESSES ARE RUNNING

Because commands can be processed in the background, it is not always obvious which processes you are responsible for. The **ps** command stands for process status and displays information about processes associated with your terminal. The **ps** command produces output such as:

```
  PID  TTY      TIME    CMD
10308   38      1:36    ed chap02.man
   49   38      0:29    -sh
11267   38      0:00    ps
```

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

To see all the processes running on the system, use the **-a** option:

```
ps -a
```

To find out about the processes running on another terminal, specify the terminal number. For example, to find out what processes are associated with terminal 13, type:

```
ps -tl3
```

For more information about **ps** and its options, see **ps(C)** in the XENIX User and System Administrator Reference Manual.

FINDING OUT LINEPRINTER INFORMATION

The **lpstat** command displays information files waiting to be printed.

To find out the status of one file, you need to know the request ID. This appears on your screen when you make print requests using the **lp** command. The request ID has the form:

printer - idnumber

printer is the name of the printer your file will be printed on (check with your system administrator for the names of printers available to you). *idnumber* is a unique number identifying your file.

To find out the status of a particular file, type:

lpstat *request_ID*

lpstat responds by displaying the date and time you made your print request and the number of characters remaining to be printed.

To find out the status of all your files waiting to be printed on the lineprinters, type:

```
lpstat
```

lpstat responds by displaying the request IDs and status information for all your files.

You can find out what files are waiting to be printed on a particular printer by using **lpstat** with the **-p** option. This command has the form:

lpstat -p *printer*

lpstat responds by printing the request IDs and status information for all the files waiting to be printed on the named printer.

For more information on **lpstat** and its options, see **lpstat(C)** in the XENIX User and System Administrator Reference Manual.

USING THE LINEPRINTER

The XENIX lineprinter commands are easy to use and give you great flexibility when you want to print a file. With a few simple commands, you can print multiple copies of a file, cancel a print request, or ask for a special option on a particular printer. Since the XENIX lineprinter system is designed to be easily adapted to many different environments, check with your system administrator to find out what lineprinters and printer options are available to you.

PRINTING FILES: lp

To print copies of your files, you can use either the **lp** or **lpr** commands. These commands are equivalent. The examples in this section use **lp**.

For example, to print one copy of a file named memo, type:

```
lp memo
```

To print multiple files, add the filenames to the command line:

```
lp memo report letter
```

When you make print requests, **lp** responds by displaying your request ID on your terminal screen. Request IDs are of the form:

```
pr4-532
```

The first part (pr4) is the name of the printer your file will be printed on. The second part (532) identifies your file. Should you later wish to cancel your print request or check its status, you will find it useful to remember your request ID.

One copy of each file you named will be printed on the default destination printer on your system.

You can use **lp** with pipes and other commands. The command to paginate a file is **pr**. To paginate and print a file named textfile, type:

```
pr textfile | lp
```

To sort, paginate, and print a file named datafile, type:

```
sort datafile | pr | lp
```

USING lp OPTIONS

The **lp** command has several options to help you control the output from your printer.

You can specify the number of copies you want printed by using the number option, **-n**. For example, to print two copies of a file named report, type:

```
lp report -n2
```

Another option, **-d** specifies your file's destination, that is, which printer your file will be printed on. Check with your system administrator for the names of the printers available to you. To have two copies of a file named report printed on a printer named quick, type:

```
lp report -n2 -dquick
```

Other useful options include:

-c Makes a copy of the files you are printing. This prevents you from inadvertently removing or changing the file before it is printed.

-m Sends you mail telling you your file has been printed.

-o Specifies printer options. For example, you may be able to request that your document be printed using 12 pitch type. Check with your system administrator to see what options are available for each printer or groups of printers on your system.

-r Removes your files after printing.

For more information on options available for the **lp** command, see **lp(C)** in the XENIX User and System Administrator Reference Manual.

CANCELING A PRINT REQUEST: cancel

You can cancel a print request. For example, to stop printing a file with a request ID of laser-245, type:

```
cancel laser-245
```

The **cancel** command immediately stops the file from being printed, even if the printer has already begun the print request.

You can also use the **cancel command** to stop whatever is currently printing on a particular printer.

For example, to cancel whatever file is currently printing on a printer named slow, type:

```
cancel slow
```

If the file did not belong to you, mail will automatically be sent to the file's owner reporting that the print request was canceled.

FINDING OUT THE STATUS OF YOUR PRINT REQUEST: lpstat

To find out the status of your files waiting to be printed, type:

```
lpstat
```

You'll receive a message such as:

```
prtl-121    chrisw    450    Dec 15 09:30
laser-450   chrisw    4968   Dec 15 09:46
```

The first column shows the request ID for each of your files being printed; the second column is your login name; the third column lists the number of characters to be printed; the fourth column lists the dates and times you made your print requests.

To learn the status of a particular file, use the **lpstat** command with your request ID. For example, to find out the status of a file with the request ID of daisy-256, type:

```
lpstat daisy-256
```

lpstat displays the status of that file only.

You can also request the status of various printers on your system by using the **-p** option or by giving the name of the particular printer you are interested in.

To find out the status of all the printers on your system, type:

```
lpstat -p
```

To find out the status of a printer named quick, type:

```
lpstat -pquick
```

lpstat displays the request ID and status information for each file currently waiting to be printed on the printer named quick.

For more information on **lpstat** and its options, see **lpstat(C)** in the XENIX User and System Administrator Reference Manual.

COMMUNICATING WITH OTHER USERS

Because the XENIX system supports multiple users, it is very convenient to communicate with other users of the system. The various methods of communication are described below. See Chapter 5, "Using mail," and **mail(C)** in the XENIX User and System Administrator Reference Manual for more information.

SENDING MAIL

mail is a system-wide facility that permits you and other system users to send and receive mail. To send mail to another user on the system, use the **mail** command as follows:

mail *username*

where *username* is the name of any user of the system. Following entry of the command, you enter the actual text of the message you want to send. Type **CTRL D** to complete your message.

RECEIVING MAIL

When you log in you may sometimes see the message:

you have mail

To read your mail, type:

mail

A heading for each message appears on your terminal screen. When you press **CR**, the contents of the first message are displayed. Subsequent messages appear one message at a time, most recent message first, each time you press **CR**

Once each message is displayed, you must decide what to do with it. The two basic responses are **d**, which deletes the message, and **CR**, which retains the message in your mailbox. To exit mail, press **q**, for "quit".

WRITING TO A TERMINAL

To write directly to another user's terminal, use the **write** command. For example, to write to joe's terminal, type:

write joe

After you have executed the command by pressing **CR**, each subsequent line that you type is displayed both on your own terminal screen and on joe's. To terminate the writing of text to joe, enter a **CTRL D** alone on a line. The procedure for a two-way write is for each party to end each message with a distinctive signal, normally (o) for "over"; when a

conversation is about to be terminated use the signal (oo) for "over and out".

USING THE SYSTEM CLOCK AND CALENDAR

Several XENIX commands will tell you the date and time, or display a calendar for any month or year you choose. The following sections explain these commands.

FINDING OUT THE DATE AND TIME

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

```
date
```

The date and time appear in the bottom left corner of the screen.

DISPLAYING A CALENDAR

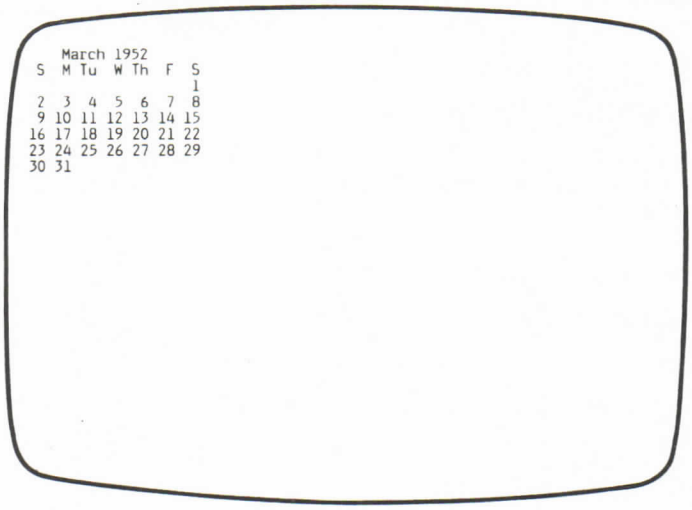
The **cal** command displays the calendar of any month or year you specify. This command has the form:

cal *month year*

For example, to display the calendar for March, 1952, type:

```
cal 3 1952
```

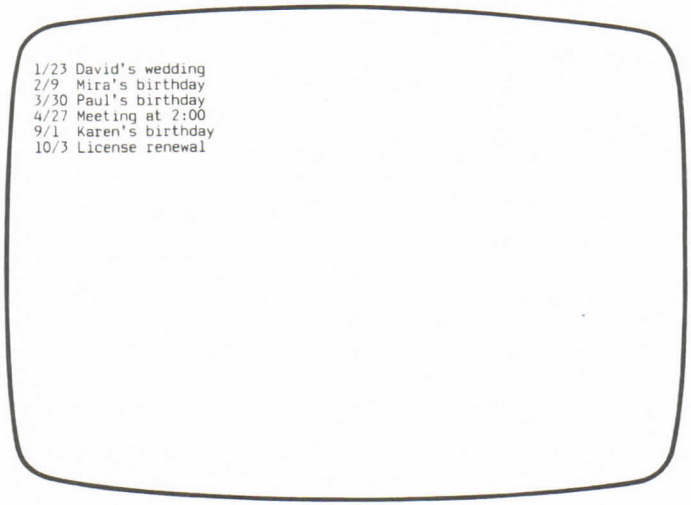
The result is:



The month must always be expressed as a digit. To display the calendar for an entire year, leave out the month. The year must always be expressed in full; the command "cal 84" displays the calendar for the year 84, not 1984.

USING THE AUTOMATIC REMINDER SERVICE

An automatic reminder service is normally available for all XENIX users. Once each day, XENIX uses the **cal** command to examine each user's home directory for a file named calendar, the contents of which might look something like this:



cal examines each line of the calendar file, extracting from the file those lines containing today's and tomorrow's dates. These lines are then mailed to you to remind you of the specified events.

USING ANOTHER USER'S ACCOUNT

You can easily access another user's files, regardless of the permission settings, with the **su** command. The **su** procedure resembles logging in, and you must know the other user's password. For example, to become user Joe, type:

```
su joe
```

and press **CR** . When the password prompt appears, type Joe's password. To cancel the effect of the **su** command and return to your own account, press **CTRL D**

CALCULATING

The **bc** command invokes an interactive desk calculator that can be used as if it were a hand-held calculator.

For more information, see Chapter 6, "bc: A Calculator".

5. USING mail

ABOUT THIS CHAPTER

This chapter serves as an introduction to the XENIX V mail system.

CONTENTS

INTRODUCTION

The XENIX mail system is a versatile communication facility that allows XENIX users to compose, send, receive, forward, and reply to mail. Users can also create distribution groups and send copies of messages to multiple users. These functions are integrated into XENIX so that all users can quickly and easily communicate with each other.

This chapter is organized to satisfy the needs of both the beginning and advanced user. The first sections discuss basic concepts, tasks, and commands. Later sections discuss advanced topics and provide quick reference to the mail program's many functions. The major sections in this chapter are:

Demonstration	Shows new users how to get started.
Basic Concepts	Discusses the fundamental ideas and terminology used in mail.
Using Mail	Shows how to perform common mailing procedures such as composing, sending, forwarding, and replying to mail.
Commands	Discusses each mail command.
Leaving Compose Mode Temporarily	Discusses and gives examples of each command available when composing a message. These commands are called compose escapes.
Setting Up Your Environment	Discusses the user's mail startup file and options that customize functions.
Using Advanced Features	Discusses advanced features such as using mail as a reminder service and handling a large volume of mail.
Quick Reference	Summarizes all commands, compose escapes, and options.

DEMONSTRATION

The **mail** command lets you perform two distinct functions: sending mail and deleting mail. In this demonstration, you'll learn to send mail to yourself, read a message, delete it, and exit the mail program.

COMPOSING AND SENDING A MESSAGE

To begin, type:

```
mail self
```

where *self* is your user name. Next, type the following lines, ending each with a **CR** :

```
This is a message sent to myself.
I compose.a message by entering lines of text.
The message is ended by typing CTRL D on a new line.
```

As you enter the message you can use compose escapes to perform special functions. To get a list of the available compose escapes, type:

```
~?
```

on a new line. To specify a subject, use the ~s escape. For example, type:

```
~s Sample subject
```

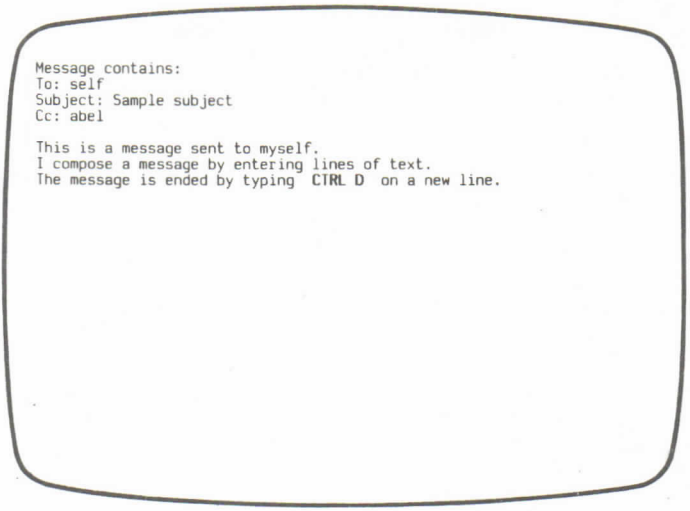
To specify a list of people to receive carbon copies use the ~c escape. For example, type:

```
~c abel
```

To view the message as it will appear when you send it, type:

```
~p
```

This will print the following:



Finally, to end the message and send it to those you have mentioned in the **To:** and the **Cc:** fields, press **CTRL D** by itself on a line. You will exit the mail program and return to the XENIX shell. Once you have sent mail, there is no way to retrieve it.

READING MAIL

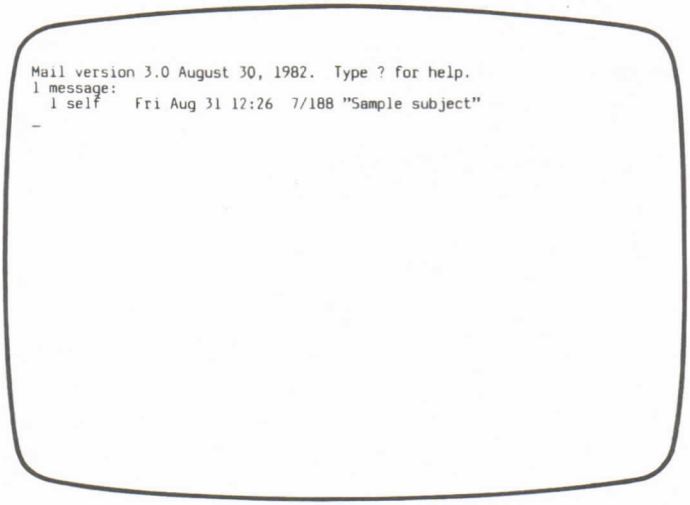
Within a short time, when you press **CR** you should receive the message:

```
You have mail.
```

This indicates that a message has arrived in your system mailbox. To read this message and any others that may have been sent to you, type:

```
mail
```

Mail then displays a sign-on message and a list of message headers that look something like this:

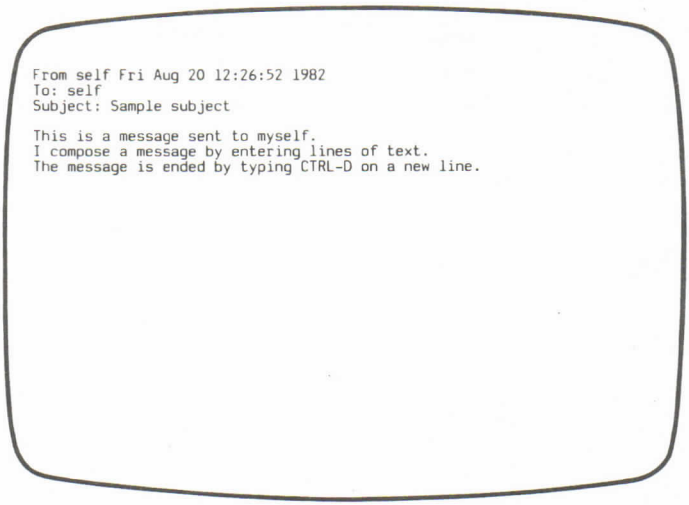


The message header indicates:

- who sent the message
- when it was sent
- the number of lines and characters
- the subject of the message

The most recent message appears at the top of the list. Messages are numbered in ascending order from least to most recent, so the most recent message has the highest number.

To see the message that you sent to yourself press **CR** . **mail** prints the following:



Note that the message you sent to yourself now contains information about the sender of the message -- a line telling who sent the message and when it was sent. The next line tells who the message was sent to. A subject and carbon copy (**Cc:**) field can be specified by the sender. If present, they too are displayed when you read the message.

DELETING A MESSAGE

To delete the message, type:

```
d
```

when the underscore appears beneath the message you want deleted.

LEAVING MAIL

To exit **mail**, type:

```
q
```

mail then displays the message:

```
0 messages held in /usr/spool/mail/self
```

and returns you to the XENIX shell.

This ends the demonstration. For more detailed information, see the discussions in following sections.

BASIC CONCEPTS

It is much easier to use **mail** if you understand its basic concepts. This section presents the following concepts:

- mailboxes
- messages
- modes
- command syntax

MAILBOXES

It is useful to think of the mail system as a typical postal system. What is normally called a post office is called the system mailbox. The system mailbox contains a file for each user in the directory /usr/spool/mail. Your own personal or user mailbox is the mbox file in your home directory. Mail sent to you is put in your system mailbox; you may choose to save mail in your user mailbox after you have read it. Note that the user mailbox differs from a real mailbox in several respects:

1. You decide whether mail is to be placed in the user mailbox; it is not automatically placed there.
2. The user mailbox is not the place where mail is initially routed -- that place is the system mailbox in the directory /usr/spool/mail.
3. Mail is not picked up from your user mailbox.

MESSAGES

In **mail**, the message in the basic unit of exchange between users. Messages consist of two parts: a heading and a body. The heading contains the following fields:

To: This field is mandatory and contains one or more user names corresponding to real users to whom you send mail.

Subject: This optional field contains text describing the message.

Cc: The carbon copy field contains one or more names of those who will receive copies of a message. Message recipients see these names in the received message. This field can be empty.

Bcc: The blind carbon copy field contains one or more names of people who will receive copies of a message. Recipients do not see these names in

the received messages. This field can be empty.

Return-receipt-to: The return receipt to: field contains the name or names of those who are to receive an automatic acknowledgement of the message. This field can be empty.

The body of a message is text exclusive of the heading. The body can be empty.

MODES

This section discusses **mail**'s modes of operation.

When you invoke **mail** you are using the shell. You can enter a message from your shell by typing:

```
mail john
```

followed by the text of your message. Press **CR** to start a new line, and **CTRL D** to send the message.

Such messages are created in mail's compose mode. When entering text in compose mode, there are several special keys associated with line editing functions: these are the same special characters that are available to you when executing normal XENIX commands. For example, you can kill the line you are editing by typing the kill character, normally a **CTRL U**. To backspace, press either **CTRL H** or the **BKSP** key.

From compose mode, you can issue commands called compose escapes. These are also called tilde escapes because the command letters are preceded by a tilde (~). When you execute these commands you are temporarily leaving or escaping from compose mode; hence the name. Note that once you've pressed **CR** to end a line, you cannot change that line from within compose mode; to change it, you must enter edit mode.

The most common way of using **mail** is to just type:

```
mail
```

This automatically places you in mail command mode. In this mode, you are prompted by an underscore for commands that permit you to manipulate your mail.

You can enter edit mode from either compose mode or command mode. In edit mode, you edit the body of a message using the full capabilities of an editor. To enter edit mode from command mode, use either the **e** or **edit** command to enter **ed** , or the **v** or **visual** command to enter **vi**. To enter edit mode from compose mode, use the compose escapes **~e** and **~v** , respectively.

MESSAGE-LISTS

Many **mail** commands take a list of messages as an argument. A message list is a list of message numbers, ranges, and names, separated by **SPACEs** or **TABs** . Message numbers may be either decimal numbers, which directly specify messages, or one of the special characters ^, ., or $, which specify the first, current, or last undeleted message, respectively.

A range of messages is two message numbers separated by a dash. To print the first four messages on the screen, type:

```
p 1-4
```

To print all the messages from the current message to the last message, type:

```
p .-$
```

A name is a user name. Messages can be printed by specifying the name of the sender. For example, to print each message sent to you by "john", type:

```
p john
```

As a shorthand notation, you can specify asterisk (*) to get all undeleted messages. For example, to print all messages except those that have been deleted, type:

```
p *
```

To delete all messages, type:

```
d *
```

To undelete all messages, type:

```
u *
```

HEADERS

When you enter **mail**, a list of message headers appears. A header is a single line of text containing descriptive information about a message. (Note that we use the word heading to describe the first part of a message, and header to describe mail's one-line description of a message.) The information includes:

- The number of the message
- The sender
- The date sent

- The number of characters and lines

- The subject (if the message contains a Subject: field)

The **headers** command lets you display up to 18 message headers. If there are more than 18 messages, the list is divided into an appropriate number of windows. To move forward one window at a time, type:

```
headers +
```

To move backward one window, type:

```
headers -
```

COMMAND SYNTAX

Each **mail** command has its own syntax. Some take no arguments, some take only one, and others take several arguments. The more flexible commands, such as **print** , accept combinations of message lists and user names. For these commands, **mail** first gathers all message numbers and ranges, then finds all messages from any specified user names. The full message list is the intersection of these two sets of messages. Thus, the message list "4-15 miller" matches all messages between 4 and 15 that are from miller.

Each **mail** command is typed on a line by itself, and any arguments follow the command word. The command need not be typed in its entirety - the first command that matches the typed prefix is used. For example, you can type **p** instead of **print** for the **print** command and **h** instead of **headers** for the **headers** command.

After the command itself is typed, one or more spaces should be entered to separate the command from its arguments. If a **mail** command does not take arguments, the arguments you give are ignored. For commands that take message lists as arguments, if no message list is given, the last message printed is used. If it does not satisfy the requirements of the command, the search proceeds forward. If there are no messages forward of the current message, the search proceeds backwards, and if there are no good messages at all, **mail** displays:

```
No applicable messages
```

USING MAIL

This section describes how to perform some basic tasks when using mail. More detailed discussions of each of these commands are presented in later sections.

ENTERING AND EXITING MAIL

To begin a **mail** session, type:

```
mail
```

The headers for each received message are then displayed one screenful at a time.

To end the **mail** session, use the **quit** (**q**) command. All messages remain in the system mailbox unless they have been deleted with the **delete** (**d**) command.

SENDING MAIL

To send a message, invoke **mail** with the names of the people and groups you want to receive the message. Next, type in your message. When you are finished, press **CTRL D** at the beginning of a line. The message is automatically sent to the specified people. While entering the text of your message, you can escape to an editor or perform other useful functions with compose escapes. The section entitled "Composing Mail" describes some features of mail that help you compose messages.

You can use redirection to mail a file that already exists. Use a command line of the form:

mail *username* < *filename*

where *filename* is the name of the file you are sending.

Note

Be very careful when mailing a file with the input redirection symbol (<). If you accidentally type the output redirection symbol (>), you will overwrite the file, destroying its contents.

If mail cannot be delivered to a specified address, you will either be notified immediately, in which case a copy of the undeliverable message is appended to the file dead.letter, or you will be notified by return mail, in which case a copy is included in the return mail message.

READING MAIL

To read messages sent to you, type:

```
mail
```

mail checks your mail out of the system mailbox and prints out a one-line header of each message, one screenful at a time. The most recent message header appears at the top of the list. You can move forward one message by pressing **CR** or typing **+** . To move forward *n* messages use **+** *n* . You can move backwards one message with the **-** command or move backwards *n* messages and print with **-** *n* . You can also move to any

arbitrary message and print it by typing its number.

If new messages arrive while you are in **mail**, the following message appears:

New mail has arrived--type 'restart' to read.

Type:

restart

and the headers of the new messages appear at the top of the list.

DISPOSING OF MAIL

After examining a message you can delete it with the **delete** command, reply to it with the **reply** (**r**) command, forward it with the **forward** (f) command, or skip to the next message by pressing **CR**

COMPOSING MAIL

To compose mail, you must enter compose mode. Do this from XENIX command level by typing:

mail *username*

where *username* is the name of a user to whom you want to send mail. From mail command mode, you can enter compose mode with the **mail** , **reply** , or **Reply** commands. Once in compose mode, the text that you type is appended one line at a time to the body of the message you are sending. Normal line editing functions are available, including **CTRL U** to kill a line and **BKSP** to back up one character. To abort a message you have begun, press **INTERRUPT** twice.

While you are composing a message, **mail** treats lines beginning with the tilde character (~) in a special way. This character introduces commands called compose escapes. The section entitled "Leaving Compose Mode Temporarily" discusses the many uses of compose escapes.

FORWARDING MAIL

To forward a message, use the **forward** (**f**) command. For example, to forward a copy of mail from your box to John's, type:

f john

The copy is shifted right one tab stop, and the new message is forwarded to John. John will receive a message heading indicating that you have forwarded the message. The **Forward** (**F**) command works just like its lowercase counterpart, except that the forwarded message is not shifted right.

REPLYING TO MAIL

You can use the **reply** command to set up a response to a message, automatically addressing a reply to the person who sent the original message. You then type in text and send the message by pressing **CTRL D** on a line by itself. The **reply** command works just like its lowercase counterpart, except that the message is sent to others named in the original message's **To:** and **Cc:** fields.

CREATING MAILING LISTS

You can create personal mailing lists so that, for example, you can send mail to cohorts and have it go to a group of people. Such lists are defined by placing an alias line like:

```
alias cohorts bill bob barry
```

in the file .mailrc in your home directory. To display the current list of such aliases use the **alias** (**a**) mail command. Personal aliases are expanded when the mail is sent. For example, the **To:** field in a message sent to cohorts will read:

```
To: bill bob barry
```

and not:

```
To: cohorts
```

Normally, system-wide aliases are available to all users. These are installed by your system administrator.

SETTING OPTIONS

Ther are several options that you can set from mail command mode or in the file .mailrc in your home directory. These options are discussed in the section entitled "Setting Up Your Environment: the .mailrc File".

COMMANDS

This section describes each of the commands available to you in mail command mode. The examples in this section assume you have invoked **mail** and that you have several messages you want to dispose of. Note that in general, **mail** commands can be invoked with either the name of the command or a one- or two-character mnemonic abbreviation.

GETTING HELP: help AND ?

The **help** (?) command prints out a brief summary of all **mail** commands, so if you ever get stuck when you are in mail command mode, type:

```
?
```

or

```
help
```

READING MAIL: p, +, -, AND restart

To look at a specific message, use the **print** (**p**) command. For example, imagine you have a header list that looks like this:

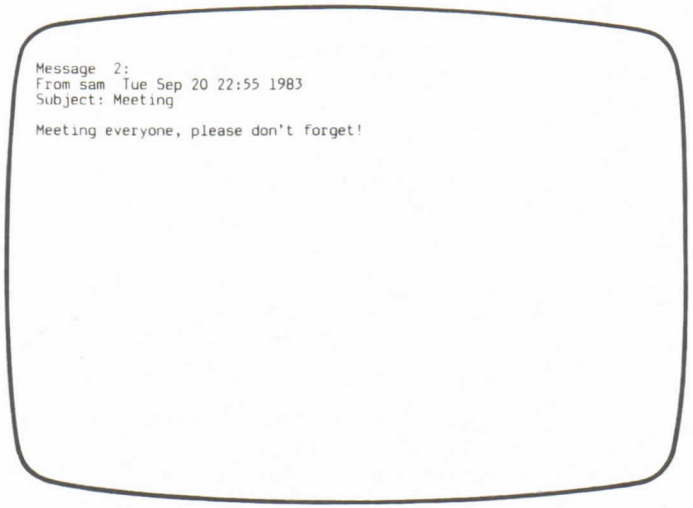
```
3 john     Wed Sep 21 09:21  26/782 "Notice"
2 sam      Tue Sep 20 22:55  6/83   "Meeting"
1 tom      Mon Sep 19 01:23  6/84   "Invite"
```

Reading from the left, each header contains the message number, who sent it, the day, date, and time it was sent, the number of lines and characters in the message, and its subject.

To examine the second message, type:

```
p 2
```

This might cause mail to respond with:



To look at message 3, type:

```
-
```

or to look at message 1, type:

```
+
```

The commands + and - execute relative to the last message referred to, which in our example was 2. For large numbers of messages, you can skip forward and backward by the number of messages specified as an argument to + and -. For example, typing:

```
+3
```

skips forward three messages. If you type:

```
p *
```

then all messages are displayed, since the star (*) matches all messages.

Pressing **CR** prints out the next message in the header list. You can always go to a message and print it by giving its message number or one of the special characters, caret (^), dot (.), or dollar sign ($). In the example where message 2 is the current message, to print the current message, type:

To print message 1, type:

```
^
```

To print message 3, type:

```
$
```

When new mail arrives while you are in mail, the message:

```
New mail has arrived - type restart to read
```

appears. If you wish to read the new messages, type:

```
restart
```

The headers of the new messages appear at the top of the list.

FINDING OUT THE NUMBER OF THE CURRENT MESSAGE: =

The **number** (=) command prints out the message number of the current message. It takes no arguments.

DISPLAYNG THE FIRST FIVE LINES: t

The **top** (**t**) command prints the first five lines of each addressed message. For example:

```
top 2-12
```

prints out the first five lines of each of the messages 2 through 12. Note that the number of lines printed out by **top** can be set with the toplines option.

DISPLAYING HEADERS: h

The **headers** (**h**) command displays up to 18 headers.

To examine the next set of 18 headers, type:

```
h +
```

To examine the previous set, type:

```
h -
```

Both plus and minus take an optional numeric argument that indicates the number of header windows to move forward or backward before printing. If a message list is given, then the **headers** command prints out the header line for each message in the list. For example:

```
h joe
```

displays all the message headers from joe. The following are some

characteristics of the header list:

- Deleted messages do not appear in the listing.
- Messages saved with the **save** command are flagged with an asterisk (*).
- Messages to be saved in your user mailbox are flagged with an "M".
- If the autombox option is set, messages held with the **hold** command are flagged with an "H".

DELETING MESSAGES: d AND dp

Unless you indicate otherwise, each message you receive is automatically saved in the system mailbox when you quit **mail**. Often, however, you don't want to save messages you have received. To delete messages, use the **delete** (**d**) command. For example:

```
d 1
```

deletes message 1 from the system mailbox.

The **dp** command deletes the current message and prints the next message. It is useful for quickly reading and disposing of mail. Using **dp** is the same as using the **d** command with the autoprint option set. See also the **undelete** command.

UNDELETING MESSAGES: u

The **undelete** (**u**) command retrieves a message previously deleted with the **d** or **dp** commands. For example, to undelete message 3, type:

```
u 3
```

You cannot undelete messages from previous mail sessions; they have been permanently deleted.

LEAVING MAIL : q AND x

To leave mail, use the **quit** (**q**) command. All messages are held in your system mailbox, except the following:

- Deleted messages, which are discarded irretrievably.
- Messages marked with the **mbox** command, which are saved in mbox in your home directory.
- Messages saved with the **save** and **write** commands, which are deleted from the system mailbox.

If you wish to leave mail quickly without altering either your system or user mailbox, use the **exit** (**x**) command. This returns you to the shell without changing anything: no messages are deleted or saved.

SAVING YOUR MAIL: s

The **save** (**s**) command lets you save messages to a specified file. The **save** command writes out each message to the file given as the last argument on the command line. For example, the following command appends messages 1-5 to the file letters:

```
s 1-5 letters
```

The file letters is created if it does not already exist. In a header list, each saved message is marked with a asterisk (*).

Save writes the entire message, including the **To:** , **Subject:** , and **Cc:** fields. In comparison, the **write** command writes only the bodies of the specified messages.

SAVING YOUR MAIL: w

The **write** (**w**) command writes the body of each message to the file given as the last argument on the command line. Each written message is marked with a asterisk (*). The syntax is similar to that of the **save** command. For example:

```
w 3-17 john elliot book
```

writes out the bodies of all messages from john and elliot in the number range 3-17. They are concatenated to the end of the file named book.

SAVING YOUR MAIL: mb

The **mbox** (**mb**) command marks each message specified in a message list, so that all are saved in the user mailbox when a **quit** command is executed. Message headers are marked with an "M" to show that they are to be saved in mbox.

SAVING YOUR MAIL: ho

The **hold** (**ho**) command takes a message list and marks each message so that it is saved in your system mailbox instead of deleted or saved in mbox when you quit. Since files in the system mailbox are saved by default, use **hold** only when you have also set the autombox option.

PRINTING YOUR MAIL ON THE LINEPRINTER: l

The **lpr** (**l**) command paginates and prints out messages to the lineprinter. It takes a message list as its argument, then paginates and prints out each message. For example:

```
l doug
```

prints out each message from the user doug on the lineprinter.

SENDING MAIL: **m**

To send mail to a user, use the **mail** (**m**) command. This sends mail in the manner described for the **reply** command, except that you supply a list of recipients either as an argument or by entering them in the **To:** field. All compose escapes work in mail. Note that the **mail** command is in most ways identical to typing mail users at the XENIX command level.

REPLYING TO MAIL: r AND R

Often, you want to respond to a message right away. The **reply** (**r**) command is useful for this purpose: it takes a message list and sends mail to the author of each message. The original message's subject field is copied as the reply's subject. Each message is composed in compose mode; thus all compose escapes work in reply, and messages are terminated by pressing **CTRL D** .

The **Reply** (**R**) command works just like its lowercase counterpart, except that copies of the reply are also sent to everyone shown in the original message's **To:** and **Cc:** fields.

FORWARDING MAIL: f AND F

To forward a copy of a message, use the **forward** (**f**) command. This sends a copy of the current message to the specified users. For example, to forward the current message to someone whose login name is john, type:

```
f john
```

John will receive the forwarded message, along with a heading showing that you are the one who forwarded it. Inside the new message, the forwarded message is indented one tab stop. An optional message number can also be given. For example:

```
f 2 john bill
```

forwards message 2 to john and bill.

The **Forward** (**F**) command is identical to the lowercase **forward** command, except that the forwarded message is not indented.

CREATING MAILING LISTS: a

The **alias** (**a**) command links a group of names with the single name given by the first argument, thus creating a mailing list. For example, you could type:

```
alias beatles john paul george ringo
```

so that whenever you used the name beatles in a destination address (as in "mail beatles"), it would expand to list the four names associated with the beatles alias. With no arguments, **alias** prints out all currently-defined aliases. With one argument, it prints out the users defined by the given alias.

You will probably want to define aliases in the startup file, .mailrc, so that you don't have to redefine them each time you invoke mail. See the section entitled "Setting Up Your Environment: The .mailrc File" for more information.

SETTING AND UNSETTING OPTIONS: se AND uns

Mail switch and string options can be set with the mail commands **set** and **unset**. A switch option is either on or off (set or unset). String options are strings of characters that are assigned values with the syntax *option* = *string* . Multiple options may be specified on a line. It is most useful to place **set** and **unset** commands in the file .mailrc in your home directory, where they become your own personal default options when you invoke **mail**. For example, you might have a **set** command that looked like this:

```
set dot metoo toplines=10 SHELL=/usr/bin/sh
```

The options **dot** and **metoo** are switch options; **toplines** and **SHELL** are string options.

The command:

```
set ?
```

prints out a list of the available options. See the section entitled "Setting up Your Environment: The .mailrc File" for descriptions of these options.

EDITING A MESSAGE: e AND v

To edit individual messages using the text editor, use the **edit** (**e**) command. It takes a message list and processes each message in turn by writing it to a temporary file. The editor, **ed** , is then automatically invoked so that you can edit the temporary file. When you finish editing the message, write the message out, then quit the editor. **mail** reads the message back into the message buffer and removes the temporary file.

It is often useful to be able to invoke either a line or visual editor, depending on the type of terminal you are using. To invoke **vi** , you can use the **visual** (v) command. The operation of the visual command is otherwise identical to that of the **edit** command.

EXECUTING SHELL COMMANDS: sh AND !

To execute a shell command without leaving **mail** , precede the command with an exclamation point. For example:

```
!date
```

prints the current date without leaving **mail**. To enter a new shell, type:

```
sh
```

To exit from this new shell and return to mail command mode, press **CTRL D** .

FINDING OUT THE NUMBER OF CHARACTERS IN A MESSAGE: si

The **size** (**si**) command prints out the number of characters in each message in a message list. For example, the command "si 1-4" might print out:

```
4:  234
3:  1000
2:  23
1:  456
```

CHANGING THE WORKING DIRECTORY: cd

The **cd** command changes the working directory to the name of the directory you give it as an argument. If no argument is given, the directory is changed to your home directory. This command works just like the normal XENIX **cd** command. (Note that exiting mail returns you to the directory from which you entered mail; thus the mail **cd** command works only within mail.) You may want to place a **cd** command in your .mailrc file so that you always begin executing mail from within the same directory.

READING COMMANDS FROM FILE: so

The **source** (**so**) command reads in mail commands from a named file. Normally, these commands are **alias** , **set** , and **unset** commands.

LEAVING COMPOSE MODE TEMPORARILY

While composing a message to be sent to others, it is often useful to print a message, invoke the text editor on a partial message, execute a shell command, or perform some other function. **mail** provides these capabilities through compose escapes (sometimes called tilde escapes) which consist of a tilde (~) at the beginning of a line, followed by a single character that specifies the function to be performed. These escapes are available only when you are composing a new message. They have no meaning when you are in mail command mode. The available compose escapes are as follows:

GETTING HELP: ~?

The help escape:

```
~?
```

prints a brief summary of the available compose escapes on your screen.

PRINTING THE MESSAGE: ~p

The print escape:

```
~p
```

prints a line of dashes and the heading and body of the message you are composing.

EDITING THE MESSAGE: ~e and ~v

The editor escapes, ~e and ~v , copy the current message into a temporary file so it can be edited. After modifying the message to your satisfaction, write it out and quit the editor. **mail** responds with the message:

```
(continue)
```

after which you may continue composing your message.

EDITING HEADERS: ~t, ~s, ~c, ~b, ~R AND ~h

To add additional names to the list of message recipients, type the escape:

t *name1 name2 ...*

You can name as many additional recipients as you wish. Note that users originally on the recipient list will still receive the message: you cannot remove anyone from the recipient list with ~t. To remove a recipient, use the ~h command, which is discussed later in this section.

You can replace or add a subject field by using the ~s escape:

~s *line-of-text*

This replaces any previous subject with *line-of-text* . The subject, if given, appears near the top of the message, prefixed with the heading **Subject** . You can see what the message looks like by using ~p, which prints out all heading fields along with the body of the text.

You may occasionally prefer to list certain people as recipients of carbon copies of a message rather than direct recipients. The escape:

~c *name1 name2 ...*

adds the named people to the **Cc:** list. The escape:

~cc *name1 name2 ...*

performs an identical function. Similarly, the escape:

~b *name1 name2 ...*

adds the named people to the **Bcc:** (Blind carbon copy) list. The people on this list receive a copy of the message, but are not mentioned anywhere in the message you send. Remember that you can always execute a ~p escape to see what the message looks like.

The escape:

~R

adds or changes the person or persons named in the **Return-receipt-to:** field.

The recipients of the message are given in the **To:** field; the subject is given in the **Subject:** field, carbon copy recipients are given in the **Cc:** field and the return receipt recipient in the **Return-receipt-to:** field. If you wish to edit these in ways impossible with the ~t, ~s, ~c and ~R escapes, you can use:

~h

where **h** stands for "heading". The escape ~h prints **To:** followed by the

current list of recipients and leaves the cursor at the end of the line. If you type in ordinary characters, they are appended to the end of the current list of recipients. You can also use the normal XENIX command line editing characters to edit these fields; you can erase existing heading text by backspacing over it.

When you press **CR** , **mail** advances to the **Subject:** field, where the same rules apply. Another **CR** brings you to the **Cc:** field, another brings you to the **Bcc:** field, and yet another to the **Return-receipt-to:** field. Each of these fields can be edited in the same way. Finally, another **CR** leaves you appending text to the end of your message body. As always, you can use ~p to print the current text of the heading fields along with the body of the message.

ADDING A FILE TO THE MESSAGE: ~r AND ~d

It is often useful to be able to include the contents of another file in your message. The escape:

~r *filename*

is provided for this purpose, and causes the named file to be appended to your current message. **mail** displays a message if the file doesn't exist or can't be read. If the read is successful, **mail** prints the number of lines and characters appended to your message.

As a special case of ~r, the escape:

~d

reads in the file dead.letter in your home directory. This is often useful because mail copies the text of your message buffer to dead.letter whenever you abort the creation of a message by either typing two consecutive interrupts or entering a ~q escape.

ENCLOSING ANOTHER MESSAGE: ~m AND ~M

If you are sending **mail** from within **mail** you can insert a message sent to you into the message you are currently composing. For example, you might type:

~m 4

This reads message 4 into the message you are composing, shifted right one tab stop. The escape:

~M 4

performs the same function, but with no right shift. You can name any nondeleted message or list of messages.

SAVING THE MESSAGE IN A FILE: ~w

To save the current text of a message body in a file, use:

~w *filename*

mail writes out the message body to the specified file, then prints the number of lines and characters written to the file. The ~w escape does not write the message heading to the file.

LEAVING MAIL TEMPORARILY: ~! AND ~|

To temporarily escape to the shell, use the escape

~! *command*

This executes *command* and returns you to mail compose mode without altering your message. If you wish to filter the body of your message through a shell command, use:

~| *command*

This pipes your message through the command and uses the output as the new text of your message. If the command produces no output, **mail** assumes that something is wrong, retains the old version of your message, and prints:

(continue)

ESCAPING TO MAIL COMMAND MODE: ~:

To temporarily escape to **mail** command mode, use either of the following escapes:

~:mail- *command*
~_mail- *command*

You can then execute any mail command that you want. Note that this escape will not work in most cases if you enter compose mode from the XENIX shell. It depends on the command used (**set** and **unset** will work), but most commands that involve message lists are not allowed. You will receive the message:

May not execute cmd while composing

PLACING A TILDE AT THE BEGINNING OF A LINE: ~~

If you wish to send a message that contains a line beginning with a tilde, you must type it twice. For example, typing:

```
~~This line begins with a tilde.
```

appends:

```
~This line begins with a tilde.
```

to your message. The escape character can be changed to a different character with the escape option. If the escape character is not a tilde, then this discussion applies to that character and not the tilde.

SETTING UP YOUR ENVIRONMENT: THE .MAILRC FILE

Whenever **mail** is invoked, it first reads the file /usr/lib/mail/mailrc then the file .mailrc in the user's home directory. System-wide aliases are defined in /usr/lib/mail/mailrc. Personal aliases and set options are defined in .mailrc . The following is a sample .mailrc file:

```
alias office bill steve karen
alias cohorts john mary bob beth mike
set dot askcc
cd
```

THE SUBJECT PROMPT: asksubject

The **asksubject** switch causes prompting for the subject of each message before you enter compose mode. If you respond to the prompt with a **CR** , no subject field is sent.

THE CC PROMPT: askcc

The **askcc** switch causes prompting for additional carbon copy recipients when you finish composing a message. Responding with a **CR** signals your satisfaction with the current list. Pressing **INTERRUPT** prints:

```
interrupt
(continue)
```

so that you can return to editing your message.

PRINTING THE NEXT MESSAGE: autoprint

The **autoprint** switch causes the **delete** command to behave like **dp**. After deleting a message, the next message in the list is automatically printed. Printing also occurs automatically after execution of an **undelete** command.

LISTING MESSAGES IN CHRONOLOGICAL ORDER: chron and mchron

The **chron** switch causes messages to be listed in chronological order. By default, messages are listed with the most recent first. Set **chron** when you want messages to appear in the order they were received.

The **mchron** switch, like **chron** , prints messages in chronological order, but lists them in the opposite order, that is, highest-numbered, or most recent, first. This is useful if you keep a large number of messages in your mailbox and you wish to list the headers of the most recently received mail first but read the messages themselves in chronological order.

USING THE PERIOD TO SEND A MESSAGE: dot

The **dot** switch lets you use a period (.) as an end-of-transmission character, as well as **CTRL D** . This option is helpful for users accustomed to this convention from editing with the editor, **ed**.

INCLUDING YOURSELF IN A GROUP: metoo

Usually, when an alias expands, the sender's name is removed from the expansion. Setting the **metoo** option causes the sender to be included in the group.

SAVING ABORTED MESSAGES: save

The **nosave** switch prevents aborted messages from being appended to the file dead.letter in your home directory; messages are saved by default. Messages are aborted from compose mode by two interrupts or a **~q** compose escape.

PRINTING THE VERSION HEADER: quiet

The **quiet** switch suppresses the printing of " *n* messages:" before the header list and suppresses printing of the version header when **mail** is first invoked.

CHOOSING AN EDITOR: THE EDITOR STRING

The EDITOR string contains the pathname of the text editor to use in the edit command and ~e escape. If not defined, then the default editor is used. For example:

```
set EDITOR=/bin/ed
```

CHOOSING AN EDITOR: THE VISUAL STRING

The VISUAL string contains the pathname of the text editor used in the visual command and ~v escape. For example:

```
set VISUAL=/bin/vi
```

By default, vi is the editor used.

CHOOSING A SHELL: THE SHELL STRING

The SHELL string contains the name of the shell to use in the ! command and the ~! escape. A default shell is used if this option is not defined. For example:

```
set SHELL=/bin/sh
```

CHANGING THE ESCAPE CHARACTER: THE ESCAPE STRING

The escape string defines the character to use in place of the tilde (~) to denote compose escapes. For example:

```
set escape=*
```

With this setting, the asterisk becomes the new compose escape character.

SETTING PAGE SIZE: THE PAGE STRING

The page string causes messages to be displayed in pages of size n lines. You are prompted with a question mark between pages. Pressing CR causes the next page of the current message to be printed. By default this paging feature is turned off.

SAVING OUTGOING MAIL: THE RECORD STRING

The **record** string sets the pathname of the file used to record all outgoing mail. If undefined, outgoing mail is not copied and saved. For example:

```
set record=/usr/john/recordfile
```

With this setting, all outgoing mail is automatically appended to the file /usr/john/recordfile.

KEEPING MAIL IN THE SYSTEM MAILBOX: autombox

The **autombox** switch determines whether messages remain in the system mailbox when you exit mail. If you set **autombox** , examined messages are automatically placed in the mbox file in your home directory (your user mailbox) and removed from the system mailbox when you quit.

CHANGING THE TOP VALUE: THE TOPLINES STRING

The **toplines** string sets the number of lines of a message to be printed out with the **top** command. By default, this value is five. For example:

```
set toplines=10
```

With this setting, ten lines of each message are printed out when the **top** command is used.

SENDING MAIL OVER TELEPHONE LINES: ignore

The **ignore** switch causes interrupt signals from your terminal to be ignored and echoed as at-signs (@). This switch is normally used only when communicating with mail over telephone lines.

USING ADVANCED FEATURES

This section discusses advanced features of **mail** useful to those familiar with the XENIX mail system.

COMMAND LINE OPTIONS

One very useful command line option to **mail** is the subject switch (**-s**). With this switch you can specify a subject on the command line. For example, you could send a file named letter with the subject line, "Important Meeting at 12:00", by typing the following:

```
mail -s "Important Meeting at 12:00" john bob mike <letter
```

To include other header fields in your message, you can use the following options:

-b *user* Adds the blind carbon copy field to the message header.

-c *user* Adds the carbon copy field to the message header.

-r *user* Adds the return-receipt to: field to the message header.

None of the above options may be specified more than once on a **mail** command line. If multiple arguments are required for an option, the entire argument set must be enclosed in quotes, as in:

```
mail -r "meeting" -b singleuser -c "x y z" user user2
```

mail also allows you to edit files of messages by using the -f switch on the command line. For example:

```
mail -f filename
```

causes **mail** to edit *filename* and the *command* :

```
mail -f
```

causes **mail** to read mbox in your home directory. All **mail** commands except hold are available to edit the messages. When you type the **quit** command, mail writes the updated file back.

If you send mail over a noisy phone line, you may notice that bad characters are transmitted. Many of these will be the character that aborts messages: the **RUBOUT** or **DEL** character. To ignore these bad characters, invoke **mail** with the -i switch.

When you enter the **mail** program (as opposed to sending a message from command level), two command line options are available:

-R Makes the **mail** session read-only, preventing alteration of the mail being read.

-u *user* Reads in user's mail instead of your own.

USING MAIL AS A REMINDER SERVICE

You can also use **mail** as a reminder service. XENIX automatically examines the file named calendar in each user's home directory and looks for lines containing either today or tomorrow's date. These lines are sent by **mail** as reminders of important events.

You can also write shell procedures to have **mail** signal the completion of a job. For information about writing shell procedures, see Chapter 7, "The Shell".

HANDLING LARGE AMOUNTS OF MAIL

You must periodically examine your user mailbox contents to decide whether messages are still relevant; you should only save important mail in your user mailbox.

One organizational method is to save mail in files organized by sender, by topic, or by a combination of the two. Create these files in a separate mail directory; you can access these mailbox files with the mail **-f** *filename* switch. However, be forewarned - this approach to organizing mail quickly eats up disk space.

MAINTENANCE AND ADMINISTRATION

The following programs and files make up the XENIX mail system:

/usr/bin/mail	Mail program
/usr/lib/mail/mailrc	Mail system initialization file
/usr/spool/mail/*	System mailbox files
/usr/name/dead.letter	File where undeliverable mail is deposited
/usr/name/mbox	User mailbox
/usr/name/.mailrc	User mail initialization file
/usr/lib/mail/mailhelp.cmd	Mail command help file
/usr/lib/mail/mailhelp.esc	Mail compose escape help file
/usr/lib/mail/mailhelp.set	Mail option help file
/usr/lib/mail/aliases	System-wide aliases
/usr/lib/mail/aliases.hash	System-wide alias database
/usr/lib/mail/faliases	Forwarding aliases
/usr/lib/mail/maliases	Machine aliases
/usr/lib/mail/maliases.hash	Optional machine aliases database

A system-wide distribution list is kept in /usr/lib/mail/aliases. A system administrator is usually in charge of this list. These aliases are kept in a vastly different syntax from .mailrc, and are expanded when mail is sent. You will normally need special permission to change system-wide aliases.

QUICK REFERENCE

The following sections provide quick reference to the available commands, compose escapes, and options.

COMMAND SUMMARY

The following are the names and syntax for each command, the abbreviated form (in brackets), and a short description. Many commands have optional arguments; most can be executed without any arguments at all. In particular, commands that take a message-list argument will default to the current message if no message list is given.

COMMAND	FUNCTION
CR	Prints the next message.
+ *n*	(+) With no *n* argument, goes to the next message and prints it. If given a numeric argument fInfR, goes to the nth message and prints it.
- *n*	(-) With no *n* argument, goes to the previous message and prints it. If given a numeric argument *n* , goes to the nth previous message and prints it.
^	Prints the first message.
$	Prints the last message.
=	Prints the message number of the current message.
?	Prints the summary of mail commands in /usr/lib/mail/mailhelp.cmd.
! *Shell-cmd*	Executes the shell command that follows. No space is needed after the exclamation point.
Alias *users*	Prints system-wide aliases for users. At least one user must be specified.

COMMAND	FUNCTION
alias *name users*	(**a**) Aliases users to name. With no name arguments, prints all currently defined aliases. With one argument, prints the users aliased by the given name argument.
cd *directory*	(**c**) Changes the user's working directory to the specified directory. If no directory is given, then changes to the user's home directory.
delete *msg-list*	(**d**) Deletes each message in the given message list.
dp *msg-list*	Deletes the current message and prints the next message.
echo *path*	Expands shell metacharacters.
edit *msg-list*	(**e**) Takes the given message list and points the text editor at each message in turn. On return to command mode, the edited message is read back in. See also the fBvisualfR command.
exit[!]	(**x**) Immediately returns to the shell without modifying the system mailbox, the user mailbox, or a file specified with the **-f** switch.
file	(**fi**) Prints the name of the mailbox file.
forward *msg-num user-list*	(**f**) Takes a *user-list* argument and forwards the current message to each name. The message sent to each is indented and shows that the sender has passed it on. The *msg-num* argument is optional, and is used to forward the numbered message instead of the default message.

COMMAND	FUNCTION
Forward *msg-num user-list*	(**F**) Same as forward except that the message is not indented.
headers + *n* |- *n* | *msg-list*	(**h**) With no argument, lists the current range of headers, which is an 18-message group. If a plus (+) argument is given, then the next 18-message group is printed, and if a minus (-) argument is given, the previous 18-message group is printed. Both plus and minus accept an optional numeric argument indicating the number of header windows to move forward or backward. If a message list is given, then the message header for each message in the list is printed.
help	Same as **?** . Prints the summary of mail commands in /usr/lib/mail/mailhelp.cmd.
hold *msg-list*	(**ho**) Takes a message list and marks each message to be saved in the user's system mailbox instead of in mbox.
list	Prints list of **mail** commands.
lpr *msg-list*	(**l**) Prints each of the messages in the required message list on the lineprinter. Messages are piped through *pr* before being printed.
mail *[user-list]*	(**m**) Takes an optional user-list argument and sends mail to each name after entering compose mode.
mbox *msg-list*	(**mb**) Marks messages given in the message-list argument to be saved in the user mailbox when a quit is executed. Message headers contain an initial letter "M" to show that they are to be saved.
move *msg-list msg-num*	Places the messages specified in *msg-list"* after the message specified in *msg-num* . If *msg-num* is 0, *msg-list* moves to the top of the mailbox.

COMMAND	FUNCTION
print *msg-list*	(**p**) Takes a message list and prints each message on the user's terminal.
quit	(**q**) Terminates the **mail** session, retaining all nondeleted, unsaved messages in the system mailbox. If the **autombox** option is set, then examined messages are saved in the user mailbox, deleted messages are discarded, and all messages marked with the **hold** command are retained in the system mailbox. If you are executing a **quit** while editing a mailbox file with the **-f** flag, the mailbox file is rewritten and the user returns to the shell.
reply *msg-list*	(**r**) Takes a message list and sends mail to each message author just like the **mail** command.
Reply *msg-list*	(**R**) Sends a reply to users named in the To: and Cc: fields, as well as the original sender.
restart	Reads in mail that arrives during the current mail session.
save *msg-list filename*	(**s**) Takes an optional message list and a filename and appends each message in turn to the end of the file. The default message is the current message.
set	(**se**) Prints list of available options.
set *option-list*	(**se**) With no arguments, prints all variable values. Otherwise, sets option. Arguments are of the form *option=value,* if the option is a string option or just *option,* if the option is a switch. Multiple options may be set on one line.
shell	(**sh**) Invokes an interactive version of the shell.
size *msg-list*	(**si**) Takes a message list and prints the size in characters of each message.

COMMAND	FUNCTION
source *file*	(**so**) Reads and executes **mail** commands from the named file.
string *string msg-list*	Searches for *string"* in *msg-list* . If no *msg-list* is specified, all undeleted messages are searched. Ignores case in search.
top	(**t**) Takes a message list and prints the top five lines. The number of lines printed is set by the variable *toplines* .
undelete *msg-list*	(**u**) Takes a message list and marks each one as not being deleted. Each message in the list must previously have been deleted.
unset *options*	(**uns**) Takes a list of option names and discards their remembered values; this is the opposite of **set** .
visual *msg-list*	(**v**) Invokes the **vi** editor on each message specified in the list.
whois	Looks up a list of target mail recipients and prints the real names or descriptions of each recipient. If the first character of the first argument is alphabetic, the arguments are looked up without change. Otherwise, the arguments are assumed to be a message list. For each message in the list, the "From" person is extracted from the header and added to list of users to be searched.
write *msg-list filename*	(**w**) Writes the message bodies of messages given by the message list to the file given by *filename*.

COMPOSE ESCAPE SUMMARY

Compose escapes are used when composing messages to perform special functions. They are only recognized at the beginning of lines. The escape character can be set with the escape string option. Abbreviations for each escape are in brackets.

Here is a summary of the compose escapes:

~~ *string*	Inserts the string of text in the message prefaced by a single tilde (~).
~?	Prints out help for compose escapes on terminal.
~.	Same as **CTRL D** on a new line.
~! *command*	Executes a shell command, then returns to compose mode.
~| *command*	Pipes the message body through the command as a filter. Replaces the message body with the output of the filter. If the command gives no output or terminates abnormally, retains the original message body.
~_ *mail-command*	Executes a mail command, then returns to compose mode.
~: *mail-command*	Executes a mail command, then returns to compose mode.
~alias	(~a) Prints list of private aliases.
~alias *aliasname*	(~a) Prints names included in private aliasname.
~alias *aliasname users*	(~a) Adds users to private aliasname list.
~Alias	(~A) Performs aliasing by first examining private aliases and then system-wide aliases using all three global alias files. Only the final result is printed (non-local mail recipients will have the complete delivery path printed). The user list is taken from header fields.
~Alias *users*	(~A) Performs aliasing by first examining private aliases and then system-wide aliases using all three global alias files. Only the final result is printed (non-local mail recipients will have the complete delivery path printed). At least one user must be specified.
~bcc *name...*	(~b) Adds the given names to the **Bcc:** field.
~cc *name...*	(~c) Adds the given name to the **Cc:** field.

~dead	(~d) Reads the file dead.letter from your home directory into the message.
~editor	(~e) Invokes the line editor on the message being sent. Exiting the editor returns the user to compose mode.
~headers	(~h) Allows the user to modify each field in the message heading.
~message *msg-list*	(~m) Reads the named messages into the message being sent, shifted right one tab. If no messages are specified, reads the current message.
~Message *msg-list*	(~M) Same as ~message except with no right shift.
~print	(~p) Prints the message buffer prefaced by the message heading.
~Print	(~P) Prints the real names or descriptions (in parentheses) after each recipient.
~quit	(~q) Aborts the message being sent, copying the message to "dead.letter" in your home directory if the save option is set.
~read *filename*	(~r) Reads the named file into the message.
~Return *name*	(~R) Adds the given names to the Return-receipt-to: field.
~shell	(~sh) Invokes a shell.
~subject *string*	(~s) Causes the named string to become the current subject field.
~to *name...*	(~t) Adds the given names to the To: field.
~visual	(~v) Invokes the vi editor to edit the message buffer. Exiting the editor returns the user to compose mode.
~write *filename*	(~w) Writes the message body to the named file.

OPTION SUMMARY

Options are controlled with the **set** and **unset** commands. An option is either a switch or a string. A switch is either on or off, while a string option has a value that is a pathname, a number, or a single character.

askcc	Causes prompting for additional carbon copy recipients at the end of each message. Pressing **CR** retains the current list.
asksubject	Causes prompting for the subject of each message you send. The subject is a line of text terminated by a **CR** .
autombox	Usually messages are retained in the system mailbox when the user quits. However, if this option is set, examined messages are automatically appended to the user mailbox.
autoprint	Causes the **delete** command to behave like **dp**. Thus, after deleting (or undeleting) a message, the next one is printed automatically.
chron	Causes messages to be listed in chronological order.
dot	Causes a single period on a new line to act as the EOT character. The normal end-of-transmission character, **CTRL D** , still works.
EDITOR=	Pathname of the text editor to use in the edit command and ~e escape. If not defined, then a default editor is used.
escape=char	If defined, sets char as the character to use in place of the tilde (~) to denote compose escapes.
ignore	Causes interrupt signals from your terminal to be ignored and echoed as at-signs (@).
mchron	Causes messages to be listed in numerical order (most recently received first), but displayed in chronological order.
metoo	Normally, before sending, the name of the sender is removed from alias expansions. If **metoo** is set, then the name of the sender is not removed.

nosave	Prevents saving of the message buffer in the file dead.letter in the home directory, after two consecutive interrupts or a ~q escape.
page= *n*	Specifies the number of lines to be printed in a "page" of text when displaying messages.
quiet	Suppresses the printing of the version when mail is first invoked.
record=	Sets the pathname of the file used to record all outgoing mail. If not defined, then outgoing mail is not copied.
SHELL=	Pathname of the shell to use in the ! command and the ~! escape. A default shell is used if this option is not defined.
toplines=	Sets the number of lines of a message to be printed with the top command. Default is five lines.
verify	Causes each target mail recipient to be verified. This option permits errors made while composing messages to be corrected or ignored.
VISUAL=	Pathname of the text editor to use in the **visual** command and ~v escape. The default is for the **vi** editor.

6. bc: A CALCULATOR

ABOUT THIS CHAPTER

This chapter serves as an introduction to the XENIX V calculator.

CONTENTS

INTRODUCTION

bc is a program that can be used as an arbitrary precision arithmetic calculator. **bc**' s output is interpreted and executed by a collection of routines which can input, output, and do arithmetic on indefinitely large integers and on scaled fixed-point numbers. Although you can write substantial programs with **bc**, it is often used as an interactive tool for performing calculator-like computations. The language supports a complete set of control structures and functions that can be defined and saved for later execution. The syntax of **bc** has been deliberately selected to agree with the C language; those who are familiar with C will find few surprises. A small collection of library functions is also available, including sin, cos, arctan, log, exponential, and Bessel functions of integer order.

Common uses for **bc** are:

- Computation with large integers
- Computations accurate to many decimal places
- Conversions of numbers from one base to another base

A scaling provision permits the use of decimal point notation. Provision is made for input and output in bases other than decimal. Numbers can be converted from decimal to octal simply by setting the output base equal to 8.

The actual limit on the number of digits that can be handled depends on the amount of storage available on the machine, so manipulation of numbers with many hundreds of digits is possible.

DEMONSTRATION

This demonstration is designed to show you:

- How to get into and out of **bc**
- How to perform simple computations
- How expressions are formed and evaluated
- How to assign values to registers

To invoke **bc**, use the command:

```
bc
```

To exit **bc**, type:

```
quit
```

or press **CTRL D** . Once you have entered **bc**, you can use it very much like a normal calculator. As with the XENIX shell, commands are read as

command lines, so each line that you type must be terminated by a **CR**. Throughout this chapter, the **CR** is implied at the end of each command line. Within **bc**, normal processing of other keys, such as **BKSP** and **INTERRUPT**, also works.

For example, type the simple integer 5:

```
5
```

Output is immediately echoed on the next line to the standard output, which is normally the terminal screen:

```
5
```

Here "5" is a simple numeric expression. However, if you type the expression:

```
5*5.25
```

(where the asterisk (*) is the multiplication operator) a computation is executed and the result printed on the next line:

```
26.25
```

What has happened here is that the line "5*5.25" has been evaluated, i.e., the expression has been reduced to its most elementary form, which is the number 26.25. The process of evaluation normally involves some type of computation such as multiplication, division, addition, or subtraction. For example, all four of these operations are involved in the following expression:

```
(10*5)+50-(50/2)
```

When this expression is evaluated, the subexpressions within parentheses are evaluated first, just as they would be with simple algebra, so that an intermediate step in the evaluation is "50+50-25" which ultimately reduces to the number "75".

The simple addition:

```
10.45+5.5555555
```

produces the output:

```
16.0055555
```

Note how precision is retained in the above result.

The two-part multiplication:

```
(8*9)*7
```

produces the answer:

```
504
```

The last part of this demonstration shows you how to store values in special alphabetic registers. For example, type:

```
a=100 ; b=5
```

What happens here is that the registers "a" and "b" are assigned the values 100 and 5, respectively. The semicolon is used here to place multiple **bc** statements on a single line, just as it is used in the XENIX shell. This command line produces no output because assignment statements are not considered expressions. However, the registers "a" and "b" can now be used in expressions. Thus you can now type:

```
a*b; a+b
```

to produce:

```
500
105
```

To exit **bc**, remember to type:

```
quit
```

or press **CTRL D** .

This ends the demonstration. Following sections describe use of **bc** in more detail. The final section of this chapter is a **bc** language reference.

TASKS

This section describes how to perform common **bc** tasks.

COMPUTING WITH INTEGERS

The simplest kind of statement is an arithmetic expression on a line by itself. For instance, if you type:

```
142857 + 285714
```

and press **CR** , **bc** responds immediately with the line:

```
428571
```

Other operators also can be used. The complete list includes:

OPERATOR	REPRESENTS
+	addition
-	subtraction
*	multiplication
/	devision
%	modulo (remaindering)
^	exponentiation

Division of integers produces an integer result truncated toward zero. Division by zero produces an error message.

Any term in an expression can be prefixed with a minus sign to indicate that it is to be negated (this is the "unary" minus sign). For example, the expression:

```
7+-3
```

is interpreted to mean that -3 is to be added to 7.

More complex expressions with several operators and with parentheses are interpreted just as in FORTRAN, with exponentiation performed first, then multiplication, division, modulo, and finally, addition, and subtraction. The contents of parentheses are evaluated before expressions outside the parentheses. All of the operations are performed from left to right, except exponentiation, which is performed from right to left. Thus the following two expressions:

```
a^b^c  and  a^(b^c)
```

are equivalent, as are the two expressions:

```
a*b*c  and  (a*b)*c
```

bc shares with FORTRAN and C the convention that a/b*c is equivalent to (a/b)*c.

Internal storage registers to hold numbers have single lowercase letter names. The value of an expression can be assigned to a register in the usual way, thus the statement:

```
x = x + 3
```

has the effect of increasing by 3 the value of the contents of the register named "x". When, as in this case, the outermost operator is the assignment operator (=), then the assignment is performed but the result is not printed. There are 26 available named storage registers, one for each letter of the alphabet.

There is also a built-in square root function whose result is truncated to an integer. For example, the lines:

```
x = sqrt(191)
x
```

produce the printed result:

```
13
```

SPECIFYING INPUT AND OUTPUT BASES

bc has special internal quantities called **ibase** and **obase.** **ibase** is initially set to 10, and determines the base used for interpreting numbers that are read by **bc.** For example, the lines:

```
ibase = 8
11
```

produce the output line:

```
9
```

and you are all set up to do octal to decimal conversions. However, beware of trying to change the input base back to decimal by typing:

```
ibase = 10
```

Because the number 10 is interpreted as octal, this statement has no effect. For those who deal in hexadecimal notation, the characters A-F are permitted in numbers (no matter what base is in effect) and are interpreted as digits having values 10-15, respectively. These characters must be uppercase and not lowercase. The statement:

```
ibase = A
```

changes you back to decimal input base no matter what the current input base is. Negative and large positive input bases are permitted; however no mechanism has been provided for the input of arbitrary numbers in bases less than 1 and greater than 16.

obase is used as the base for output numbers. The value of **obase** is initially set to a decimal 10. The lines:

```
obase = 16
1000
```

produce the output line:

```
3E8
```

This is interpreted as a three-digit hexadecimal number. Very large output bases are permitted. For example, large numbers can be output in groups of five digits by setting obase to 100000. Even strange output bases, such as negative bases, and 1 and 0, are handled correctly.

Very large numbers are split across lines with seventy characters per line. A split line that continues on the next line ends with a backslash (\). Decimal output conversion is fast, but output of very large numbers (i.e., more than 100 digits) with other bases is rather slow.

Remember that **ibase** and **obase** do not affect the course of internal computation or the evaluation of expressions; they affect only input and output conversion.

SCALING QUANTITIES

A special internal quantity called **scale** is used to determine the scale of calculated quantities. Numbers can have up to 99 decimal digits after the decimal point. This fractional part is retained in further computations. We refer to the number of digits after the decimal point of a number as its scale.

When two scaled numbers are combined by means of one of the arithmetic operations, the result has a scale determined by the following rules:

Addition,subtraction — The scale of the result is the larger of the scales of the two operands. There is never any truncation of the result.

Multiplication — The scale of the result is never less than the maximum of the two scales of the operands, never more than the sum of the scales of the operands, and subject to those two restrictions, the scale of the result is set equal to the contents of the internal quantity, **scale**.

Division — The scale of a quotient is the contents of the internal quantity, **scale**.

Modulo — The scale of a remainder is the sum of the scales of the quotient and the divisor.

Exponentiation — The result of an exponentiation is scaled as if the implied multiplications were performed. An

exponent must be an integer.

Square Root — The scale of a square root is set to the maximum of the scale of the argument and the contents of **scale**.

All of the internal operations are actually carried out in terms of integers, with digits being discarded when necessary. In every case where digits are discarded, truncation is performed without rounding.

The contents of **scale** must be no greater than 99 and no less than 0. It is initially set to 0.

The internal quantities **scale**, **ibase**, and **obase** can be used in expressions just like other variables. The line:

```
scale = scale + 1
```

increases the value of **scale** by one, and the line:

```
scale
```

causes the current value of **scale** to be printed.

The value of **scale** retains its meaning as a number of decimal digits to be retained in internal computation even when **ibase** or **obase** are not equal to 10. The internal computations (which are still conducted in decimal, regardless of the bases) are performed to the specified number of decimal digits, never hexadecimal or octal or any other kind of digits.

USING FUNCTIONS

The name of a function is a single lowercase letter. Function names are permitted to use the same letters as simple variable names. Twenty-six different defined functions are permitted in addition to the twenty-six variable names. The line:

```
define a(x){
```

begins the definition of a function with one argument. This line must be followed by one or more statements, which make up the body of the function, ending with a right brace (}). Return of control from a function occurs when a return statement is executed or when the end of the function is reached. The return statement can take either of the two forms:

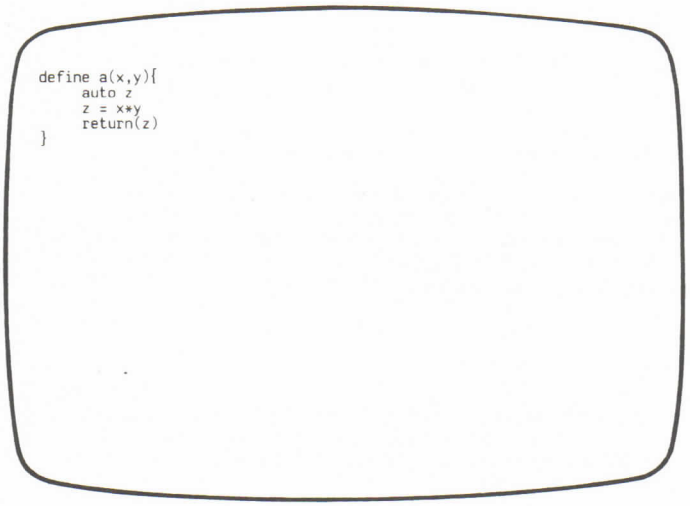
```
return
return(x)
```

In the first case, the returned value of the function is 0; in the second, it is the value of the expression in parentheses.

Variables used in functions can be declared as automatic by a statement of the form:

```
auto x,y,z
```

There can be only one **auto** statement in a function and it must be the first statement in the definition. These automatic variables are allocated space and initialized to zero on entry to the function and thrown away on return. The values of any variables with the same names outside the function are not disturbed. Functions can be called recursively and the automatic variables at each call level are protected. The parameters named in a function definition are treated in the same way as the automatic variables of that function, with the single exception that they are given a value on entry to the function. An example of a function definition follows:



The value of this function, when called, will be the product of its two arguments.

A function is called by the appearance of its name, followed by a string of arguments enclosed in parentheses and separated by commas. The result is unpredictable if the wrong number of arguments are used.

If the function "a" is defined as shown in the previous example, then the line:

```
a(7,3.14)
```

would print the result:

```
21.98
```

Similarly, the line:

```
x = a(a(3,4),5)
```

would cause the value of "x" to become 60.

Functions can require no arguments, but still perform some useful operation or return a useful result. Such functions are defined and called using parentheses with nothing between them. For example:

```
b ()
```

calls the function named b.

USING SUBSCRIPTED VARIABLES

A single lowercase letter variable name followed by an expression in brackets is called a subscripted variable and indicates an array element. The variable name is the name of the array and the expression in brackets is called the subscript. Only one-dimensional arrays are permitted in **bc**. The names of arrays are permitted to collide with the names of simple variables and function names. Any fractional part of a subscript is discarded before use. Subscripts must be greater than or equal to zero and less than or equal to 2047.

Subscripted variables can be freely used in expressions, in function calls and in return statements.

An array name can be used as an argument to a function, as in:

```
f(a[ ])
```

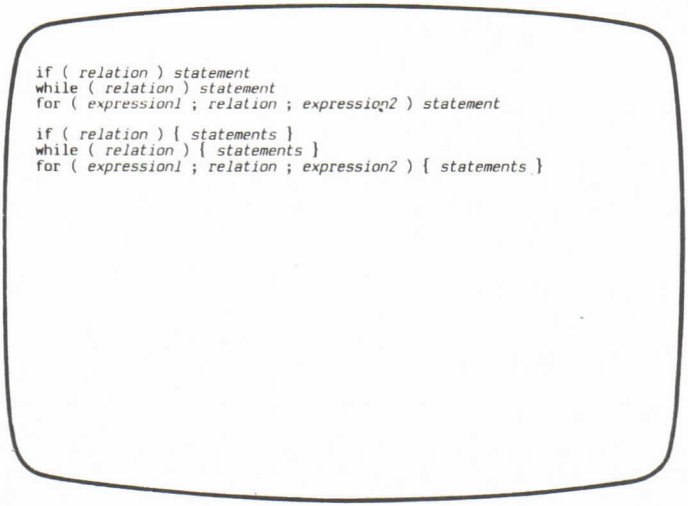
Array names can also be declared as automatic in a function definition with the use of empty brackets:

```
define f(a[ ])
auto a[ ]
```

When an array name is so used, the entire contents of the array are copied for the use of the function, then thrown away on exit from the function. Array names that refer to whole arrays cannot be used in any other context.

USING CONTROL STATEMENTS: if, while AND for

The **if**, **while**, and **for** statements are used to alter the flow within programs or to cause iteration. The range of each of these statements is a following statement or compound statement consisting of a collection of statements enclosed in braces. They are written as follows:



A relation in one of the control statements is an expression of the form:

expression1 rel-op expression2

where the two expressions are related by one of the six relational operators:

< less than

> greater than

<= less than/equal to

>= greater than/equal to

== equal to

!= not equal to

Beware of using a single equal sign (=) instead of the double equal sign (==) in a relational. Both of these symbols are legal, so you will not get a diagnostic message. However, the operation will not perform the intended comparison.

The **if** statement causes execution of its range if and only if the relation is true. Then control passes to the next statement in the sequence.

The **while** statement causes repeated execution of its range, as long as the relation is true. The relation is tested before each execution of its range, and if the relation is false, control passes to the next statement

beyond the range of the **while** statement.

The **for** statement begins by executing *expression1* . Then the relation is tested and, if true, the statements in the range of the for statement are executed. Then *expression2* is executed. The relation is tested, and so on. The typical use of the **for** statement is for a controlled iteration, as in the statement:

```
for(i=1; i<=10; i=i+1) i
```

which will print the integers from 1 to 10.

The following are some examples of the use of the control statements:

```
define f(n){
     auto i, x
     x=1
     for(i=1; i<=n; i=i+1) x=x*i
     return(x)
}
```

The line:

```
f(a)
```

prints "a" factorial if "a" is a positive integer.

The following is the definition of a function that computes values of the binomial coefficient ("m" and "n" are assumed to be positive integers):

```
define b(n,m){
     auto x, j
     x=1
     for(j=1; j<=m; j=j+1) x=x*(n-j+1)/j
     return(x)
}
```

The following function computes values of the exponential function by summing the appropriate series without regard to possible truncation errors:

```
scale = 20
define e(x){
     auto a, b, c, d, n
     a = 1
     b = 1
     c = 1
     d = 0
     n = 1
     while(1==1){
               a = a*x
          b = b*n
          c = c + a/b
          n = n + 1
          if(c==d) return(c)
          d = c
     }
}
```

USING OTHER LANGUAGE FEATURES

Every user should know the following language features:

- Statements are commonly typed one to a line. You can type several statements on a line if they are separated by semicolons.

- If an assignment statement is placed in parentheses, it then has a value and can be used anywhere that an expression can. For example, the line:

```
(x=y+17)
```

  not only makes the indicated assignment, but also prints the resulting value.

  The following is an example of a use of the value of an assignment statement even when it is not placed in parentheses:

```
x = a[i=i+1]
```

  This causes a value to be assigned to "x" and also increments "i" before it is used as a subscript.

- Following are the **bc** constructions and their equivalents:

CONSTRUCTION	EQUIVALENT
x=y=z	x =(y=z)
x =+ y	x = x+y
x =- y	x = x-y
x =* y	x = x*y
x =/ y	x = x/y
x =% y	x = x%y
x =^ y	x = x^y
x++	(x=x+1)-1
x--	(x=x-1)+1
++x	x = x+1
--x	x = x-1

If you type one of these constructions inadvertently, something legal but unexpected may happen. Be aware that in some of these

constructions spaces are significant. There is a real difference between "x=-y" and "x= -y" . The first replaces "x" by "x-y" and the second by "-y".

- The comment convention is identical to the C comment convention. Comments begin with "/*" and end with "*/".

- You can obtain a library of math functions when you invoke **bc** by typing:

  bc -l

  This command loads the library functions sine, cosine, arctangent, natural logarithm, exponential, and Bessel functions of integer order. These are named "s", "c", "a", "l", "e", and "j(n,x)", respectively.

- If you type:

  bc *file...*

  bc will read and execute the named file or files before accepting commands from the keyboard. In this way, you can load your own programs and function definitions.

LANGUAGE REFERENCE

This section is a comprehensive reference to the **bc** language. It contains a more concise description of the features mentioned in earlier sections.

TOKENS

Tokens are keywords, identifiers, constants, operators, and separators. Token separators can be spaces, tabs or comments. Newline characters or semicolons separate statements.

Comments

Comments are introduced by the characters "/*" and are terminated by "*/".

Identifiers

There are three kinds of identifiers: ordinary identifiers, array identifiers and function identifiers. All three types consist of single lowercase letters:

Ordinary identifiers are named expressions which represent a place where the current value of the expression is stored. Ordinary identifiers have no dependencies or arguments.

Array identifiers are followed by square brackets, enclosing an optional expression describing a subscript. Arrays are singly dimensioned and can contain up to 2048 elements. Indexing begins at 0,

so an array can be indexed from 0 to 2047. Subscripts are truncated to integers.

Function identifiers are followed by parentheses, enclosing optional arguments.

The three types of identifiers do not conflict; a program can have a variable named "x", an array named "x", and a function named "x", all of which are separate and distinct.

Keywords

The following are reserved keywords:

```
auto     obase
break    return
define   scale
for      sqrt
ibase    quit
if       while
length
```

Constants

Constants are arbitrarily long numbers with an optional decimal point. The hexadecimal digits A-F are also recognized as digits with decimal values 10-15, respectively.

EXPRESSIONS

All expressions can be evaluated to a value. The value of an expression is always printed unless the main operator is an assignment. The precedence of expressions (i.e., the order in which they are evaluated) is as follows:

Function calls
Unary operators
Multiplicative operators
Additive operators
Assignment operators
Relational operators

There are several types of expressions:

Named expressions

Places where values are stored. Simply stated, named expressions are legal on the left side of an assignment. The value of a named expression is the value stored in the place named.

- Ordinary identifiers are named expressions. They have an initial value of zero.

- Array elements are named expressions. They have an initial value of zero.

- The internal registers **scale**, **ibase**, and **obase** are all named expressions. **scale** is the number of digits after the decimal point to be retained in arithmetic operations and has an initial value of zero. **ibase** and **obase** are the input and output number radixes respectively. Both **ibase** and **obase** have initial values of 10.

Constants Primitive expressions that evaluate to themselves..

Parenthetic Expressions An expression surrounded by parentheses is a primitive expression. The parentheses are used to alter normal operator precedence.

Function Calls Expressions that return values. They are discussed in the next section.

FUNCTION CALLS

A function call consists of a function name followed by parentheses containing a comma-separated list of expressions, which are the function arguments. The syntax is as follows:

function-name ([*expression* [, *expression...*]])

A whole array passed as an argument is specified by the array name followed by empty square brackets. All function arguments are passed by value. As a result, changes made to the formal parameters have no effect on the actual arguments. If the function terminates by executing a return statement, the value of the function is the value of the expression in the parentheses of the return statement, or 0 if no expression is provided or if there is no return statement. There are three built-in functions:

sqrt (*expr*) The result is the square root of the expression. It is truncated in the least significant decimal place. The scale of the result is the scale of the expression or the value of **scale**, whichever is larger.

length (*expr*) The result is the total number of significant decimal digits in the expression. The scale of the result is zero.

scale (*expr*) The result is the scale of the expression. The scale of the result is zero.

UNARY OPERATORS

The unary operators bind right to left.

- expr — The result is the negative of the expression.

++ named_expr — The named expression is incremented by one. The result is the value of the named expression after incrementing.

-- named_expr — The named expression is decremented by one. The result is the value of the named expression after decrementing.

named_expr ++ — The named expression is incremented by one. The result is the value of the named expression before incrementing.

named_expr -- — The named expression is decremented by one. The result is the value of the named expression before decrementing.

MULTIPLICATIVE OPERATORS

The multiplicative operators (*, /, and %) bind from left to right.

*expr * expr* — The result is the product of the two expressions. If "a" and "b" are the scales of the two expressions, then the scale of the result is:

```
min(a+b,max(scale,a,b))
```

expr / expr — The result is the quotient of the two expressions. The scale of the result is the value of **scale**.

expr % expr — The modulo operator (%) produces the remainder of the division of the two expressions. More precisely, a%b is a-a/b*b. The scale of the result is the sum of the scale of the divisor and the value of **scale**.

expr ^ expr — The exponentiation operator binds right to left. The result is the first expression raised to the power of the second expression. The second expression must be an integer. If "a" is the scale of the left expression and "b" is the absolute value of the right expression, then the scale of the result is:

```
min(a*b,max(scale, a))
```

ADDITIVE OPERATORS

The additive operators bind left to right.

expr + *expr* — The result is the sum of the two expressions. The scale of the result is the maximum of the scales of the expressions.

expr - *expr* — The result is the difference of the two expressions. The scale of the result is the maximum of the scales of the expressions.

ASSIGNMENT OPERATORS

The following assignment operators assign values to the named expression on the left side:

named_expr = *expr* — Assigns the value of the expression on the right to the named expression on the left.

named_expr =+ *expr* — Result equivalent to *named_expr* = *named_expr* + *expr* .

named_expr =- *expr* — Result equivalent to *named_expr* = *named_expr* - *expr* .

named_expr =* *expr* — Result equivalent to *named_expr* = *named_expr* * *expr* .

named_expr =/ *expr* — Result equivalent to *named_expr* = *named_expr* / *expr* .

named_expr =% *expr* — Result equivalent to *named_expr* = *named_expr* % *expr* .

named_expr =^ *expr* — Result equivalent to *named_expr* = *named_expr* ^ *expr* .

RELATIONAL OPERATORS

Unlike all other operators, the relational operators are valid only as the object of an **if** or **while** statement, or inside a **for** statement. The relational operators are:

expr < *expr*
expr > *expr*
expr <= *expr*
expr >= *expr*
expr == *expr*
expr != *expr*

STORAGE CLASSES

There are only two storage classes in **bc**: global and automatic (local). Only identifiers that are to be local to a function need be declared with the **auto** command. The arguments to a function are local to the function. All other identifiers are assumed to be global and available to all functions.

All identifiers, global and local, have initial values of zero. Identifiers declared as auto are allocated on entry to the function and released on returning from the function: they do not retain values between function calls. Note that auto arrays are specified by the array namer, followed by empty square brackets.

Automatic variables in bc do not work the way they do in C. On entry to a function, the old values of the names that appear as parameters and as automatic variables are pushed onto a stack. Until return is made from the function, reference to these names refers only to the new values.

STATEMENTS

Statements must be separated by a semicolon or a newline. Except where altered by control statements, execution is sequential. There are four types of statements: expression statements, compound statements, quoted string statements, and built-in statements:

Expression statements When a statement is an expression, unless the main operator is an assignment, the value of the expression is printed, followed by a newline character.

Compound statements Statements grouped together and surrounded with curly braces ({ and }).

Quoted string statements For example:

```
"string"
```

prints the string inside the quotation marks.

Built-in statements Built-in statements include **auto, break, define, for, if, quit, return,** and **while.**

- The **auto** statement causes the values of the identifiers to be pushed down. The identifiers can be ordinary identifiers or array identifiers. Array identifiers are specified by following the array name by empty square brackets. The **auto** statement must be the first statement in a function definition. Syntax of the **auto** statement is:

  auto *identifier* [, *identifier*]

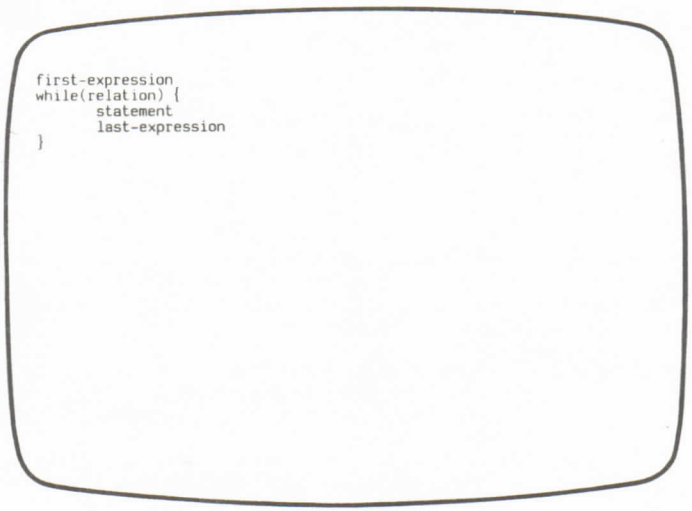
- The **break** statement terminates a **for** or **while** statement. Syntax for the break statement is:

break

- The **define** statement defines a function; parameters to the function can be ordinary identifiers or array names. Array names must be followed by empty square brackets. The syntax of the **define** statement is:

  define ([*parameter* [, *parameter...*]]){ *statements* }

- The **for** statement is the same as:



All three expressions must be present. Syntax of the **for** statement is:

for(*expression* ; *relation* ; *expression*) *statement*

- The **if** statement is executed if the relation is true. The syntax is:

  if (*relation*) *statement*

- The **quit** statement stops execution of a bc program and returns control to XENIX when it is first encountered. Because it is not treated as an executable statement, it cannot be used in a function definition or in an **if**, **for**, or **while** statement. Note that entering a **CTRL D** at the keyboard is the same as typing "quit". The syntax of the quit statement is:

  quit

- The **return** statement terminates a function, pops its auto variables off the stack, and specifies the result of the function. The result of the function is the result of the expression in parentheses. The first form is equivalent to "return(0)". The syntax of the **return** statement is:

  return(*expr* **)**

- The **while** statement is executed while the relation is true. The test occurs before each execution of the statement. The syntax of the **while** statement is:

  while (*relation* **)** *statement*

7. THE BOURNE SHELL

ABOUT THIS CHAPTER

This chapter serves as an introduction to the standard XENIX V command interface, the Bourne shell.

CONTENTS

INTRODUCTION

When users log into XENIX, they communicate with a shell command interpreter. This interpreter is a XENIX program that supports a very powerful command language. Each invocation of this interpreter is called a shell; and each shell has one function: to read and execute commands from its standard input.

Because the shell gives the user a high-level language in which to communicate with the operating system, XENIX can perform tasks unheard of in less sophisticated operating systems. Commands that would normally have to be written in a traditional programming language can be written with just a few lines in a shell procedure. In other operating systems, commands are executed in strict sequence. With XENIX and the shell, commands can be:

- Combined to form new commands
- Passed positional parameters
- Added or renamed by the user
- Executed within loops or executed conditionally
- Created for local execution without fear of name conflict with other user commands
- Executed in the background without interrupting a session at a terminal

Furthermore, commands can redirect command input from one source to another and redirect command output to a file, terminal, printer, or other command. This provides flexibility in tailoring a task for a particular purpose.

This chapter describes the Bourne shell (**sh**), which is the standard XENIX shell command interpreter. The next two chapters describe alternatives to this standard, the Visual Shell and the C-Shell.

The shell that you enter at login-time depends on how your system administrator has set up your account. You can always enter another available shell by entering its name on the command line.

BASIC CONCEPTS

The Bourne shell itself (i.e., the program that reads your commands when you log in or that is invoked with the **sh** command) is a program written in the C language; it is not part of the operating system proper, but an ordinary user program.

HOW SHELLS ARE CREATED

In XENIX, a process is an executing entity complete with instructions, data, input, and output. All processes have lives of their own, and may even start (or fork) new processes. Thus, at any given moment several processes may be executing, some of which are "children" of other processes.

Users log into the operating system and are assigned a shell from which they execute. This shell is a personal copy of the shell command interpreter that is reading commands from the keyboard: in this context, the shell is simply another process.

In the XENIX multitasking environment, files may be created in one phase and then sent off to be processed in the background. This allows the user to continue working while programs are running.

COMMANDS

The most common way of using the shell is by typing simple commands at your keyboard. A simple command is any sequence of arguments separated by spaces or tabs. The first argument specifies the name of the command to be executed. Any remaining arguments, with a few exceptions, are considered as arguments to that command. For example, the following command line requests printing of the files allan, barry, and calvin:

```
lpr allan barry calvin
```

If the first argument of a command names a file that is a compiled, executable program, the shell, as parent, creates a child process that immediately executes that program. If the file is executable, but not compiled, it is assumed to be a shell procedure, i.e., a file of ordinary text containing shell command lines. In this case, the shell spawns another instance of itself (a subshell) to read the file and execute the commands inside it.

From the user's viewpoint, compiled programs and shell procedures are invoked in exactly the same way. The shell determines which implementation has been used, rather than requiring the user to do so.

HOW THE SHELL FINDS COMMANDS

The shell normally searches for commands in three distinct locations in the file system:

- in the current directory
- in the directory /bin
- in the /usr/bin directory

For example, the **pr** and **man** commands are actually the files /bin/pr and /usr/bin/man, respectively. A more complex pathname may be given, either

to locate a file relative to the user's current directory, or to access a command with an absolute pathname. If the command name begins with a slash (/) (e.g., /bin/sort or /cmd), the prepending is not performed. Instead, a single attempt is made to execute the command as named.

This mechanism gives the user a convenient way to execute public commands and commands in or near the current directory, as well as the ability to execute any accessible command, regardless of its location in the file structure. Because the current directory is usually searched first, anyone can possess a private version of a public command without affecting other users. Similarly, the creation of a new public command does not affect a user who already has a private command with the same name. To change the sequence of directories searched, reset the shell PATH variable. (Shell variables are discussed later in this chapter.)

USING METACHARACTERS ON COMMAND LINES

Command arguments are very often filenames. Sometimes, a group of related files have similar, but not identical, filenames. The shell lets you use a special set of characters, called metacharacters, to easily specify a group of similarly named files as a command argument.

You can substitute a metacharacter for a portion of a command line, and XENIX will find all files that match the search specification. XENIX uses the following metacharacters:

METACHARACTER	MATCHES
*	any string (including the null string)
?	any one character
[...]	any character enclosed in the brackets
[x-y]	any character in the range enclosed in the brackets

For example:

*	Matches all names in the current directory
temp	Matches all names containing "temp"
[a-f]*	Matches all names beginning with "a" through "f"
*.c	Matches all names ending in ".c"
/usr/bin/?	Matches all single-character names in /usr/bin

Pattern matching has some restrictions. If the first character of a filename is a period (.), it can be matched only by an argument that

literally begins with a period. If a pattern does not match any filenames, then the pattern itself is printed out as the result of the match.

Note that directory names should not contain the following characters:

```
* ? [ ]
```

If these characters are used, then infinite recursion may occur during pattern matching attempts.

QUOTING MECHANISMS

The characters <, >, *, ?, [and] have special meanings to the shell. To remove the special meaning of these characters requires some form of quoting. This is done by using single, close quotation marks (') or double quotation marks (") to surround a string. (Single, open quotation marks (`) are used only for command substitution in the shell and do not hide the special meanings of any characters.)

All characters within single quotation marks are interpreted literally. Thus:

```
echo$tuff='echo $? $*; ls * | wc'
```

assigns the string:

```
echo $? $*; ls * | wc
```

to the variable echostuff, but does not result in any other commands being executed.

Within double quotation marks, certain characters retain their special meaning, while all other characters are interpreted literally. The characters that retain their special meaning are the dollar sign ($), the backslash (\), the single, close quotation mark ('), and the double quotation mark (") itself. Thus, within double quotation marks, variables are expanded and command substitution takes place (both topics are discussed in later sections). However, any commands in a command substitution are unaffected by double quotation marks, so that characters such as star (*) retain their special meaning.

To hide the special meaning of the dollar sign ($) and single and double quotation marks within double quotation marks, precede these characters with a backslash (\). Outside of double quotation marks, preceding a character with a backslash is equivalent to placing single quotation marks around that character. A backslash (\) followed by a newline causes that newline to be ignored and is equivalent to a space. The backslash-newline pair is therefore useful in allowing continuation of long command lines.

The following list shows how the shell interprets quoted material:

INPUT	SHELL INTERPRETS AS...
?'`'	The single, open quotation mark (`)
*	The double quotation mark (")
"`""	The double quotation mark (")
"`echo one`"	the one word "one"
"`"	illegal (expects another `)
one two	the two words "one" & "two"
"one two"	the one word "one two"
*	the one word "one two"
"one * two"	the one word "one * two"
echo one	the one word "one"

REDIRECTING INPUT AND OUTPUT

In general, most commands do not know or care whether their input or output is coming from or going to a terminal or a file. Thus, a command can be used conveniently either at a terminal or in a pipeline. A few commands vary their actions depending on the nature of their input or output, either for efficiency, or to avoid useless actions (such as attempting random access I/O on a terminal or a pipe).

When a command begins execution, it assumes that three files are already open. Each file is associated with a number, called a file descriptor:

FILE	FILE DESCRIPTOR
Standard Input	0
Standard Output	1
Diagnostic Output (or Standard Error)	2

A child process normally inherits these files from its parent. All three files are initially connected to the terminal (0 to the keyboard, 1 and 2 to the terminal screen). The shell permits the files to be redirected elsewhere before control is passed to an invoked command.

STANDARD INPUT AND OUTPUT

An argument to the shell of the form *file* or > *file* opens the specified file as the standard input or output (in the case of output, destroying the previous contents of file, if any). An argument of the form >> *file* directs the standard output to the end of *file* , thus providing a way to append data to the file without destroying its existing contents. In either of the two output cases, the shell creates file if it does not already exist. Thus:

```
>output
```

alone on a line creates a zero-length file. The following appends to file log the list of users who are currently logged on:

```
who >> log
```

Such redirection arguments are only subject to variable and command substitution; neither blank interpretation nor pattern matching of filenames occurs after these substitutions. This means that:

```
echo 'this is a test' > *.gal
```

produces a one-line file named *.gal. Similarly, an error message is produced by the following command, unless you have a file with the name "?":

```
cat < ?
```

So remember, special characters are not expanded in redirection arguments. This is so that redirection arguments are scanned by the shell before pattern recognition and expansion takes place.

DIAGNOSTIC AND OTHER OUTPUTS

Diagnostic output from XENIX commands is normally directed to the file associated with file descriptor 2. (There is often a need for an error output file that is different from standard output so that error messages do not get lost down pipelines.) You can redirect this error output to a file by immediately prepending the number of the file descriptor (2 in this case) to either output redirection symbol (> or >>). The following line appends error messages from the cc command to the file named ERRORS:

```
cc testfile.c 2>>ERRORS
```

There can be no spaces or tabs between the file descriptor number and the redirection symbol, or the number will be passed as a command argument.

You can redirect output associated with any of the first ten file descriptors (numbered 0-9). For instance, if **cmd** puts output on file descriptor 9, then the following line will direct that output to the file savedata:

```
cmd 9>savedata
```

A command often generates standard output and error output, and might even have some other output, perhaps a data file. In this case, one can redirect independently all the different outputs. Suppose, for example, that **cmd** directs its standard output to file descriptor 1, its error output to file descriptor 2, and builds a data file on file descriptor 9. The following would direct each of these three outputs to a different file:

```
cmd >standard   2>error   9>data
```

COMMAND LINES AND PIPELINES

A sequence of commands separated by the vertical bar (|) makes up a pipeline. In a pipeline consisting of more than one command, each command is run as a separate process connected to its neighbors by pipes, that is, the output of each command (except the last one) becomes the input of the next command in line.

A filter is a command that reads its standard input, transforms it in some way, then writes it as its standard output. A pipeline normally consists of a series of filters. Although the processes in a pipeline are permitted to execute in parallel, each program needs to read the output of its predecessor. Many commands operate on individual lines of text, reading a line, processing it, writing it out, and looping back for more input. Some must read large amounts of data before producing output; **sort** is an example of the extreme case that requires all input to be read before any output is produced.

The following is an example of a typical pipeline:

```
nroff -mm text | col | lpr
```

nroff is a text formatter in the XENIX Text Processing System whose output may contain reverse line motions, **col** converts these motions to a form that can be printed on a terminal lacking reverse-motion capability, and **lpr** does the actual printing. The flag **-mm** indicates one of the commonly used formatting options, and "text" is the name of the file to be formatted.

The following examples illustrate the variety of effects that can be obtained by combining a few commands in the described manner:

- who
  Prints the list of logged-in users on the terminal screen.

- who>>log
  Appends the list of logged-in users to the end of file *log*

- who | wc B -l
  Prints the number of logged-in users.

- who | pr
  Prints a paginated list of logged-in users.

- who | sort
  Prints an alphabetized list of logged-in users.

- who | grep bob
  Prints the list of logged-in users whose login names contain the string ``bob''.

- who | grep bob | sort | pr
  Prints an alphabetized, paginated list of logged-in users whose login names contain the string bob.

- { date; who | wc -l ; } >>log
  Appends (to file log) the current date, followed by the count of logged-in users. Be sure to place a space after the left brace and a semicolon before the right brace.

- who | sed -e 's/ .*//' | sort | uniq -d
  Prints only the login names of all users who are logged in more than once. Note the use of sed as a filter to remove characters trailing the login name from each line. (The ``.*'' in the **sed** command is preceded by a space.)

The **who** command does not by itself provide options to yield all these results. They are obtained by combining **who** with other commands. Note that **who** just serves as the data source in these examples. As an exercise, replace "who |" with "</etc/passwd" in the examples to see how a file can be used as a data source in the same way. Notice that redirection arguments may appear anywhere on the command line, even at the start. This means that:

< *infile* > *outfile* sort|pr

is the same as:

sort|pr< *infile* > *outfile*

COMMAND SUBSTITUTION

Any command line can be placed within single, open quotation marks (`...`) so that the output of the command replaces the quoted command line itself. This concept is known as "command substitution" . The command or commands enclosed between single, open quotation marks are first executed by the shell and then their output replaces the whole expression, single, open quotation marks and all. This feature is often used to assign to shell variables. (Shell variables are described later in this chapter.) For example:

```
today=`date`
```

assigns the string representing the current date to the variable "today"; for example "Tue Nov 27 16:01:09 EST 1982". The following command saves the number of logged-in users in the shell variable users:

```
users=`who | wc -l`
```

Any command that writes to the standard output can be enclosed in single, open quotation marks. Single, open quotation marks may be nested, but the inside sets must be escaped with backslashes (). For example:

```
logmsg=`echo Your login directory is `pwd``
```

will display the line:

your login directory is *name of login directory*

Shell variables can also be given values indirectly by using the **read** and **line** commands. The **read** command takes a line from the standard input (usually your terminal) and assigns consecutive words on that line to any variables named.

For example:

```
read first init last
```

takes an input line of the form:

```
G. A. Snyder
```

and has the same effect as typing:

```
first=G.   init=A.   last=Snyder
```

The **read** command assigns any excess "words" to the last variable.

The **line** command reads a line of input from the standard input and then echoes it to the standard output.

SHELL VARIABLES

The shell has several mechanisms for creating variables. A variable is a name representing a string value. Certain variables are referred to as positional parameters; these are the variables that are normally set only on the command line. Other shell variables are simply names to which the user or the shell itself may assign string values.

POSITIONAL PARAMETERS

When a shell procedure is invoked, the shell implicitly creates positional parameters. The name of the shell procedure itself in position zero on the command line is assigned to the positional parameter $0. The first command argument is called $1, and so on. You can use the **shift** command to access arguments in positions numbered higher than nine. For example, the following shell script might be used to cycle through command line switches and then process all succeeding files:

```
while test '$1'
   do case $1 in
   -a) A=aoption ; shift ;;
   -b) B=boption ; shift ;;
   -c) C=coption ; shift ;;
   -*) echo "bad option" ; exit 1 ;;
   *) process rest of files
   esac
done
```

One can explicitly force values into these positional parameters by using the **set** command. For example:

```
set abc def ghi
```

assigns the string "abc" to the first positional parameter, $1, the string "def" to $2, and the string "ghi" to $3. Note that $0 may not be assigned a value in this way-it always refers to the name of the shell procedure; or in the login shell, to the name of the shell.

USER-DEFINED VARIABLES

The shell also recognizes alphanumeric variables to which string values may be assigned. A simple assignment has the syntax:

name = *string*

Thereafter, $ *name* will yield the value of *string* . A *name* is a sequence of letters, digits, and underscores that begins with a letter or an underscore. No spaces surround the equal sign (=) in an assignment statement. Note that positional parameters may not appear on the left side of an assignment statement; they can be set only as described in the previous section.

More than one assignment may appear in an assignment statement, but beware: the shell performs the assignments from right to left. Thus, the following command line results in the variable "A" acquiring the value "abc":

```
A=$B B=abc
```

The following are examples of simple assignments. Double quotation marks around the right-hand side allow spaces, tabs, semicolons, and newlines to be included in a string, and allows variable substitution (also known as parameter substitution) to occur. This means that references to positional parameters and other variable names that are prefixed by a dollar sign ($) are replaced by the corresponding values, if any. Single quotation marks inhibit variable substitution.

```
MAIL=/usr/mail/gas
echovar="echo $1 $2 $3 $4"
stars=*****
asterisks='$stars'
```

In the above example, the variable "echovar" has as its value the string consisting of the values of the first four positional parameters, separated by spaces, plus the string "echo". No quotation marks are needed around the string of asterisks being assigned to stars because pattern matching (expansion of asterisk, the question mark, and brackets) does not apply in this context. Note that the value of $asterisks is the literal string "$stars", not the string "*****", because the single quotation marks inhibit substitution.

In assignments, spaces are not reinterpreted after variable substitution. The following example results in $first and $second having the same value:

```
first='a string with embedded spaces'
second=$first
```

In accessing the values of variables, you can enclose the variable name in braces {...} to delimit the variable name from any following string. In particular, if the character immediately following the name is a letter, digit, or underscore, then the braces are required. For example, examine the following input:

```
a='This is a string'
echo "${a}ent test of variables."
```

Here, the **echo** command prints:

```
This is a stringent test of variables.
```

If no braces were used, the shell would substitute a null value for "$aent" and print:

```
test of variables.
```

The following variables are maintained by the shell. Some of them are set by the shell, and all of them can be reset by the user.

HOME Specifies the name of the user's login directory, that is, the directory that becomes current following login. This variable is initialized by the login program. **cd** , without arguments, switches to the $HOME directory. Using this variable helps keep full pathnames out of shell procedures. This is of great benefit when pathnames are changed, either to balance disk loads or to reflect administrative changes.

IFS Specifies which characters are internal field separators. These are the characters the shell uses during blank interpretation. (If you want to parse some delimiter-separated data easily, you can set **IFS** to include that delimiter.) The shell initially sets **IFS** to include the blank, tab, and newline characters.

MAIL The pathname of a file where your mail is deposited. If **MAIL** is set, then the shell checks to see if anything has been added to the file it names and announces the arrival of new mail each time you return to command level (e.g., by leaving the editor). MAIL must be set by the user and exported. The **export** command is discussed later in this chapter. (The presence of mail in the standard mail file is also announced at login, regardless of whether MAIL is set.)

PATH Specifies the search path used by the shell in finding commands. Its value is an ordered list of directory pathnames separated by colons. The shell initializes **PATH** to the list :/bin:/usr/bin where a null argument appears in front of the first colon. A null anywhere in the path list represents the current directory. On some systems, a search of the current directory is not the default and the PATH variable is initialized instead to /bin:/usr/bin. If you wish to search your current directory last rather than first, use:

```
PATH=/bin:/usr/bin::
```

Here, the two colons together represent a colon followed by a null, followed by a colon, thus naming the current directory. You could possess a personal directory of commands (say, $HOME/bin) and cause it to be searched before the other three

directories by using:

```
PATH=$HOME/bin::/bin:/usr/bin
```

"PATH" is normally set in your ".profile" file.

PS1 Specifies the primary prompt string. with the value of PS1 when it expects input. The default value of PS1 is "$ " (a dollar sign ($) followed by a blank).

PS2 Specifies the secondary prompt string. If the shell expects more input when it encounters a newline in its input, it prompts with the value of PS2 . The default value for this variable is "> " (a greater-than symbol followed by a space).

In general, you should be sure to export all of the above variables so that their values are passed to all shells created from your login. Use **export** at the end of your .profile file, for example:

```
export HOME IFS MAIL PATH PS1 PS2
```

PREDEFINED SPECIAL VARIABLES

Several variables have special meanings; the following are set only by the shell:

$#** Records the number of arguments passed to the shell, not counting the name of the shell procedure itself. For instance, **$# yields the number of the highest set positional parameter. Thus:

```
sh cmd a b c
```

automatically sets **$#** to 3. One of its primary uses is in checking for the presence of the required number of arguments:

```
if test $# -lt 2
then
   echo 'two or more args required'; exit
fi
```

$?** Contains the exit status (also referred to as "return code", "exit code", or "value") of the last command executed. Its value is a decimal string. Most XENIX commands return zero to indicate successful completion. The shell itself returns the current value of **$? as its exit status.

$$ The process number of the current process. Because process numbers are unique among all existing processes, this string is often used to generate unique names for temporary files. XENIX provides no mechanism for the automatic creation and deletion of temporary files; a file exists until it is explicitly removed. Temporary files are generally undesirable; the XENIX

pipe mechanism is far superior for many applications. However, the need for uniquely-named temporary files does occasionally occur.

The following example illustrates the recommended practice of creating temporary files. Note that the directories /tmp and /usr/tmp are cleared out if the system is rebooted.

```
#       use current process id
#       to form unique temp file
temp=/usr/tmp/$$
ls > $tmp
#       commands here, some of which use $temp
rm $tmp
#       clean up at end
```

$! The process number of the last process run in the background (using the ampersand (&)). This is a string containing from one to five digits.

$-** A string consisting of names of execution flags currently turned on in the shell. For example, **$- might have the value "xv" if you are tracing your output.

THE SHELL STATE

The state of a shell is determined by the values of positional parameters, user-defined variables, environment variables, modes of execution, and the current working directory. The state may be altered in various ways, such as changing the working directory with the **cd** command, setting several flags, and by reading commands from the special file, .profile, in your login directory.

CHANGING DIRECTORIES

The **cd** command changes the current directory to the one specified as its argument. You can place **cd** within parentheses to cause a subshell to change to a different directory and execute some commands without affecting the original shell.

For example, the first sequence below copies the file /etc/passwd to /usr/you/passwd; the second example first changes directory to /etc and then copies the file:

```
cp /etc/passwd /usr/you/passwd
(cd /etc ; cp passwd /usr/you/passwd)
```

Note the use of parentheses. Both command lines have the same effect.

THE .PROFILE FILE

The file named .profile is read each time you log in to XENIX. It is normally used to execute special one-time-only commands and to set and export variables to all later shells. Only after commands are read and executed from .profile does the shell read commands from the standard input (usually the terminal).

EXECUTION FLAGS

The **set** command lets you alter the behavior of the shell by setting certain shell flags. In particular, the -x and -v flags may be useful when invoking the shell as a command from the terminal. To set flags -x and -v , type:

```
set -xv
```

To turn off the same flags, type:

```
set +xv
```

These two flags have the following functions:

-v Prints input lines as they are read by the shell. This flag is particularly useful for isolating syntax errors. The commands on each input line are executed after that input line is printed.

-x Prints commands and their arguments as they execute. (Shell control commands, such as for, while, etc., are not printed, however.) Note that -x causes a trace of only those commands that are actually executed, whereas -v prints each line of input until a syntax error is detected.

The **set** command is also used to set these and other flags within shell procedures.

A COMMAND'S ENVIRONMENT

All variables and their associated values that are known to a command at the beginning of its execution make up its environment. This environment includes variables that the command inherits from its parent process and variables specified as keyword parameters on the command line that invokes the command.

The variables that a shell passes to its child processes are those that have been named as arguments to the **export** command. The **export** command places the named variables in the environments of both the shell and all its future child processes.

Keyword parameters are variable-value pairs that appear in the form of assignments, normally before the procedure name on a command line. Such variables are placed in the environment of the procedure being invoked. For example:

```
#       keycommand
echo $a $b
```

This is a simple procedure that echoes the values of two variables. If it is invoked as:

```
a=key1 b=key2 keycommand
```

then the resulting output is:

```
key1 key2
```

Keyword parameters are not counted as arguments to the procedure and do not affect **$#** .

A procedure may access the value of any variable in its environment. However, if changes are made to the value of a variable, these changes are not reflected in the environment; they are local to the procedure in question. In order for these changes to be placed in the environment that the procedure passes to *its* child processes, the variable must be named as an argument to the **export** command within that procedure. To obtain a list of variables that have been made exportable from the current shell, type:

```
export
```

You will also receive a list of variables that have been made **readonly** . To get a list of name-value pairs in the current environment, type either:

```
printenv
```

or:

```
env
```

INVOKING THE SHELL

The shell is a command and may be invoked in the same way as any other command:

sh *proc [arg...]* A new instance of the shell is explicitly invoked to read proc. Arguments, if any, can be manipulated.

sh -v *proc [arg...]* This is equivalent to putting "set -v" at the beginning of proc. It can be used in the same way for the **-x** , **-e** , **-u** ,and **-n** flags.

proc [arg...] If *proc* is an executable file, and is not a compiled executable program, the effect is similar to that of

sh *proc args*

An advantage of this form is that variables exported in the shell will still be exported from *proc* when this form is used (because the shell only forks to read commands from *proc*). Thus any changes made within proc to the values of exported variables will be passed on to subsequent commands invoked from *proc*.

PASSING ARGUMENTS TO SHELL PROCEDURES

When a command line is scanned, any character sequence of the form **$** *n* is replaced by the nth argument to the shell, counting the name of the shell procedure itself as **$0**. This notation permits direct reference to the procedure name and to as many as nine positional parameters. You can process additional arguments with the **shift** command or **for** loop.

The shift command shifts arguments to the left; i.e., the value of **$1** is thrown away, **$2** replaces **$1** , **$3** replaces **$2**, and so on. The highest-numbered positional parameter becomes unset (**$0** is never shifted). For example, in the following shell procedure, ripple, **echo** writes its arguments to the standard output:

```
#       ripple command
while test $# != 0
do
     echo $1 $2 $3 $4 $5 $6 $7 $8 $9
     shift
done
```

Lines that begin with a number sign (#) are comments. The looping command, **while**, is discussed in the next section of this chapter. If the procedure were invoked with:

```
ripple a b c
```

it would print:

```
a b c
b c
c
```

The special shell variable, star (**$***), causes substitution of all positional parameters except **$0** . Thus, the **echo** line in the ripple example could be written more compactly as:

```
echo $*
```

These two **echo** commands are not equivalent: the first prints at most nine positional parameters; the second prints all of the current positional parameters. The shell star variable (**$***) is more concise and less error-prone. One obvious application is in passing an arbitrary number of arguments to a command. For example:

```
wc $*
```

counts the words in each file named on the command line.

It is important to understand the sequence of actions used by the shell in scanning command lines and substituting arguments. The shell first reads input up to a newline or semicolon, and then parses that much of

the input. Variables are replaced by their values and command substitution (via single, open quotation marks) is attempted. I/O redirection arguments are detected, acted upon, and deleted from the command line. Next, the shell scans the resulting command line for internal field separators, that is, for any characters specified by **IFS** to break the command line into distinct arguments; explicit null arguments (specified by "" or '') are retained, while implicit null arguments resulting from evaluation of variables that are null or not set are removed. Then filename generation occurs with all metacharacters being expanded. The resulting command line is then executed by the shell.

Sometimes command lines are built within a shell procedure. In this case, it is sometimes useful to have the shell rescan the command line after all the initial substitutions and expansions have been performed. The special command **eval** is available for this purpose. **eval** takes a command line as its argument and simply rescans the line, performing the specified variable or command substitutions. Consider the following (simplified) situation:

```
command=who
output=' | wc -l'
eval $command $output
```

This segment of code results in the execution of the command line:

```
who | wc -l
```

The output of **eval** cannot be redirected. However, uses of **eval** can be nested so that a command line can be evaluated several times.

CONTROLLING THE FLOW OF CONTROL

Several shell commands implement a variety of control structures useful in controlling the flow of control in shell procedures. Before describing these structures, a few terms need to be defined.

A simple command is any single irreducible command specified by the name of an executable file. I/O redirection arguments can appear in a simple command line and are passed to the shell, not to the command.

A command is a simple command or any of the shell control commands described in this section.

A pipeline is a sequence of one or more commands separated by vertical bars (|). In a pipeline, the standard output of each command but the last is connected (by a pipe) to the standard input of the next command. Each command in a pipeline is run separately; the shell waits for the last command to finish. The exit status of a pipeline is nonzero if the exit status of either the first or last process in the pipeline is nonzero.

A command list is a sequence of one or more pipelines separated by a semicolon (;), an ampersand (&), an "and-if" symbol (&&), or an "or-if"

(||) symbol, and optionally terminated by a semicolon or an ampersand. A semicolon causes sequential execution of the previous pipeline. This means that the shell waits for the pipeline to finish before reading the next pipeline. On the other hand, the ampersand (&) causes asynchronous background execution of the preceding pipeline. Thus, both sequential and background execution are allowed. A background pipeline continues execution until it terminates voluntarily, or until its processes are killed.

Other uses of the ampersand include off-line printing, background compilation, and generation of jobs to be sent to other computers. For example, if you type:

```
nohup cc prog.c&
```

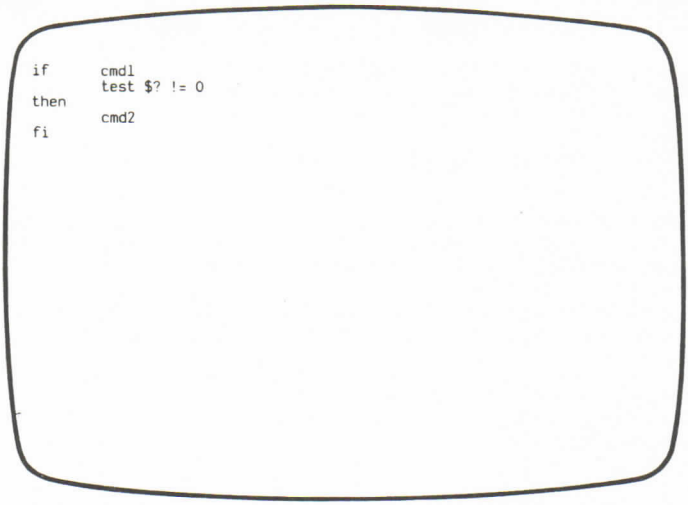
you may continue working while the C compiler runs in the background. A command line ending with an ampersand is immune to interrupts or quits that you might generate by typing **INTERRUPT** or **QUIT** . It is also immune to logouts with **CTRL D** . However, **CTRL D** will abort the command if you are operating over a dial-up line. In this case, it is wise to make the command immune to hang-ups (i.e., logouts) as well. The **nohup** command is used for this purpose. Using the previous example, if you log out from a dial-up line while cc is still executing, cc will be killed and your output will disappear.

Use the ampersand operator with restraint, especially on heavily-loaded systems.

The and-if and or-if (&& and ||) operators cause conditional execution of pipelines. Both of these are of equal precedence when evaluating command lines (but both are of lower precedence than the ampersand (&) and the vertical bar (|)). In the command line:

cmd1 || *cmd2*

the first command, *cmd1* , is executed and its exit status examined. Only if *cmd1* fails (i.e., has a nonzero exit status) is *cmd2* executed. Thus, this is a more terse notation for:



The and-if operator (&&) operator yields a complementary test. For example, in the following command line:

cmd1 && *cmd2*

the second command is executed only if the first succeeds (and has a zero exit status). In the following sequence, each command is executed in order until one fails:

cmd1 && *cmd2* && *cmd3* && ... && *cmdn*

A simple command in a pipeline may be replaced by a command list enclosed in either parentheses or braces. The output of all the commands so enclosed is combined into one stream that becomes the input to the next command in the pipeline. The following line formats and prints two separate documents:

```
{ nroff -mm text1; nroff -mm text2; } | lpr
```

Note that a space is needed after the left brace and that a semicolon should appear before the right brace.

USING THE IF STATEMENT

The shell provides structured conditional capability with the **if** command. The simplest **if** command has the following form:

```
if command-list
then command-list
fi
```

The command list following the **if** is executed and if the last command in the list has a zero exit status, then the command list that follows then is executed. The word **fi** indicates the end of the command.

To cause an alternative set of commands to be executed when there is a nonzero exit status, an **else** clause can be given with the following structure:

```
if command-list
then command-list
else command-list
fi
```

Multiple tests can be achieved in an **if** command by using the **elif** clause, although the **case** statement (see the next section) is better for large numbers of tests. For example:

```
if      test -f "$1"
#                       is $1 a file?
then    pr $1
elif    test -d "$1"
#                       else, is $1 a directory?
then    (cd $1; pr *)
else    echo $1 is neither a file nor a directory
fi
```

This example is executed as follows: if the value of the first positional parameter is a filename (**-f**), then print that file; if not, then check to see if it is the name of a directory (**-d**). If so, change to that directory (**cd**) and print all the files there (**pr***). Otherwise, **echo** the error message.

The **if** command may be nested (but be sure to end each one with a **fi**). The new lines in the above examples of **if** may be replaced by semicolons.

The exit status of the **if** command is the exit status of the last command executed in any **then** clause or **else** clause. If no such command was executed, **if** returns a zero exit status.

Note that an alternate notation for the test command uses brackets to enclose the expression being tested. For example, the previous example might have been written as follows:

```
case string in
      pattern ) command-list ;;
      ... pattern ) command-list ;;
esac
```

Note that a space after the left bracket and one before the right bracket are essential in this form of the syntax.

USING THE CASE STATEMENT

A multiple test conditional is provided by the **case** command. The basic format of the case statement is:

```
case string in
        pattern)command-list;;
        ...
        pattern)command-list;;
esac
```

The shell tries to match string against each pattern in turn, using the same pattern-matching conventions as in filename generation. If a match is found, the command list following the matched pattern is executed; the double semicolon (;;) serves as a break out of the case and is required after each command list except the last. Note that only one pattern is ever matched, and that matches are attempted in order, so that if a

asterisk (*) is the first pattern in a case, no other patterns are looked at.

More than one pattern may be associated with a given command list by specifying alternate patterns separated by vertical bars (|):

```
case $i in
    *.c)    cc $i
            ;;
    *.h | *.sh)
            : do nothing
            ;;
    *)      echo "$i of unknown type"
            ;;
esac
```

In the above example, no action is taken for the second set of patterns because the null, colon (:) command is specified. The asterisk (*) is used as a default pattern, because it matches any word.

The exit status of **case** is the exit status of the last command executed in the **case** command. If no commands are executed, then **case** has a zero exit status.

CONDITIONAL LOOPING: WHILE AND UNTIL

A **while** command has the general form:

while *command-list*
do
    command-list
done

The commands in the first command-list are executed, and if the exit status of the last command in that list is zero, then the commands in the second command-list are executed. This sequence is repeated as long as the exit status of the first command-list is zero. A loop can be executed as long as the first command-list returns a nonzero exit status by replacing **while** with **until** .

Any newline in the above example may be replaced by a semicolon. The exit status of a **while** (or **until**) command is the exit status of the last command executed in the second command-list. If no such command is executed, **while** (or **until**) has a zero exit status.

LOOPING OVER A LIST: for

To perform a set of operations for each file in a set of files, or execute a command once for each of several arguments, use the **for** command. The **for** command has the format:

```
for variable in word-list
do
        command-list
done
```

Here *word-list* is a list of strings separated by spaces. The commands in the *command-list* are executed once for each word in the *word-list* . *variable* takes on as its value each word from the *word-list* , in turn. The *word-list* is fixed after it is evaluated the first time. For example, the following for loop causes each of the C source files xec.c, cmd.c, and word.c in the current directory to be compared with a file of the same name in the directory /usr/src/cmd/sh:

```
for CFILE in xec cmd word
do      diff ${CFILE}.c /usr/src/cmd/sh/${CFILE}.c
done
```

Note that the first occurrence of CFILE immediately after the word **for** has no preceding dollar sign, since the name of the variable is wanted and not its value.

You can omit the **in** *word-list* part of a **for** command; this causes the current set of positional parameters to be used in place of *word-list*. This is useful when writing a command that performs the same set of commands for each of an unknown number of arguments. Create a file named echo2 that contains the following shell script:

```
for word
do echo $word$word
done
```

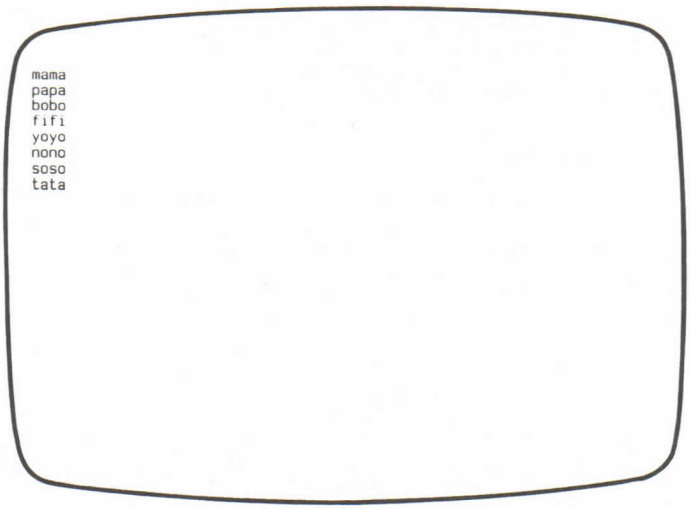
Give echo2 execute status:

```
chmod +x echo2
```

Now type the following command:

```
echo2 ma pa bo fi yo no so ta
```

The output from this command is:



LOOP CONTROL: BREAK AND CONTINUE

You can use the **break** command to terminate execution of a **while** or a **for** loop. You can use **continue** to request the execution of the next iteration of the loop. These commands are effective only when they appear between **do** and **done** .

The **break** command terminates execution of the smallest (i.e., innermost) enclosing loop, causing execution to resume after the nearest following unmatched **done** . Exit from *n* levels is obtained by **break** *n* .

The **continue** command causes execution to resume at the nearest enclosing **for** , **while** , or **until** statement (i.e., the one that begins the innermost loop containing the continue). You can also specify an argument *n* to **continue** and execution will resume at the *n* th enclosing loop:

```
# This procedure is interactive.
# "Break" and "continue" commands are used
# to allow the user to control data entry.
while true #loop forever
do      echo "Please enter data"
        read response
        case "$response" in
        "done") break
                # no more data
                ;;
        "")     # just a carriage return,
                # keep on going
                continue
                ;;
        *)      # process the data here
                ;;
        esac
done
```

END-OF-FILE AND EXIT

When the shell reaches the end-of-file in a shell procedure, it terminates execution, returning to its parent the exit status of the last command executed prior to the end-of-file. The top level shell is terminated by typing a **CTRL D** , which is equivalent to logging out.

The **exit** command simply reads to the end-of-file and returns, setting the exit status to the value of its argument, if any. Thus, a procedure can be terminated normally by placing "exit " at the end of the file.

COMMAND GROUPING: PARENTHESES AND BRACES

There are two methods for grouping commands in the shell: parentheses and braces. Parentheses cause the shell to create a subshell that reads the enclosed commands. Both the right and left parentheses are recognized wherever they appear in a command linethey can appear as literal parentheses only when enclosed in quotation marks. For example, if you type:

```
garble(stuff)
```

the shell prints an error message. Quoted lines, such as:

```
garble"("stuff")"
"garble(stuff)"
```

are interpreted correctly.

Command grouping is useful when you want to perform operations but not affect the values of current shell variables. It also lets you temporarily change the working directory and execute commands in the new directory without having to return to the current directory.

The current environment is passed to the subshell and variables that are exported in the current shell are also exported in the subshell. Thus:

```
CURRENTDIR=`pwd`; cd /usr/docs/otherdir;
nohup nroff doc.n | lpr& ; cd $CURRENTDIR
```

and:

```
(cd /usr/docs/otherdir; nohup nroff doc.n | lpr&)
```

accomplish the same result: a copy of /usr/docs/otherdir/doc.n is sent to the lineprinter. (**nroff** is a command available in the XENIX Text Processing System.) However, the second example automatically puts you back in your original working directory. In the second example, spaces or newlines surrounding the parentheses are allowed but not necessary. When entering a command line at your terminal, the shell will prompt with the value of the shell variable PS2 if an end parenthesis is expected.

Braces ({ and }) can also group commands together. Both the left and the right brace are recognized only if they appear as the first (unquoted) word of a command. The opening brace may be followed by a newline (in which case the shell prompts for more input). Unlike parentheses, no subshell is created for braces: the enclosed commands are simply read by the shell. The braces are convenient when you wish to use the (sequential) output of several commands as input to one command.

The exit status of a set of commands grouped by either parentheses or braces is the exit status of the last enclosed executed command.

INPUT/OUTPUT REDIRECTION AND CONTROL COMMANDS

The shell normally does not fork and create a new shell when it recognizes the control commands (other than parentheses) described in the previous section. However, each command in a pipeline is run as a separate process in order to direct input to or output from each command. Also, when redirection of input or output is specified explicitly to a control command, a separate process is spawned to execute that command. Thus, when **if**, **while**, **until**, **case**, and **for** are used in a pipeline consisting of more than one command, the shell forks and a subshell runs the control command. This has two implications:

1. Any changes made to variables within the control command are not effective once that control command finishes (this is similar to the effect of using parentheses to group commands).

2. Control commands run slightly slower when redirected because of the additional overhead of creating a shell for the control command.

TRANSFER TO ANOTHER FILE AND BACK: THE DOT (.) COMMAND

A command line of the form:

.proc

causes the shell to read commands from *proc* without spawning a new process. Changes made to variables in *proc* are in effect after the **dot** command finishes. This is a good way to gather a number of shell variable initializations into one file. A common use of this command is to reinitialize the top level shell by reading the .profile file with:

```
.profile
```

INTERRUPT HANDLING: trap

Shell procedures can use the command to disable a signal (cause it to be ignored), or redefine its action. The form of the **trap** command is:

trap *arg signal-list*

arg is a string to be interpreted as a command list and *signal-list* consists of one or more signal numbers as described in **signal** (S) in the XENIX User and System Administrator Reference Manual. The most important of these signals are:

NUMBER	SIGNAL
00	KILL (**CTRL U**)
01	HANGUP
02	INTERRUPT character
03	QUIT
19	KILL (cannot be caught or ignored)
11	segmentation violation (cannot be caught or ignored)
15	software termination signal

The commands in *arg* are scanned at least once, when the shell first encounters the **trap** command. Because of this, it is usually wise to surround these commands with single rather than double quotation marks. The former inhibit immediate command and variable substitution. This becomes important, for instance, when you want to remove temporary files when the names of those files have not yet been determined when the **trap** command is first read by the shell. The following procedure will print the name of the current directory in the file errdirect when it is

interrupted, thus giving the user information as to how much of the job was done:

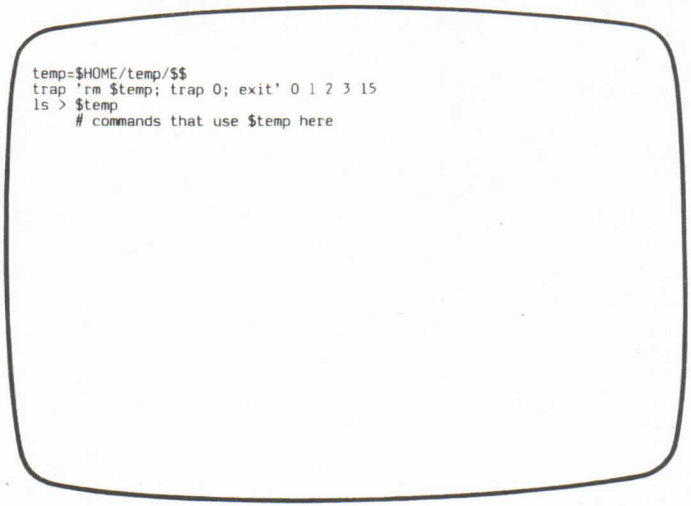
```
trap 'echo `pwd` >errdirect' 2 3 15
for i in /bin /usr/bin /usr/gas/bin
do
     cd $i
        # commands to be executed in directory $i here
done
```

Beware that the same procedure with double rather than single quotation marks does something different. The following prints the name of the directory from which the procedure was first executed:

```
(trap "echo `pwd` >errdirect" 2 3 15)
```

A signal 11 can never be trapped, because the shell itself needs to catch it to deal with memory allocation. Zero is interpreted by the **trap** command as a signal generated by exiting from a shell. This occurs either with an **exit** command, or by "falling through" to the end of a procedure. If *arg* is not specified, then the action taken upon receipt of any of the signals in the signal list is reset to the default system action. If *arg* is an explicit null string ('' or ""), then the signals in the signal list are ignored by the shell.

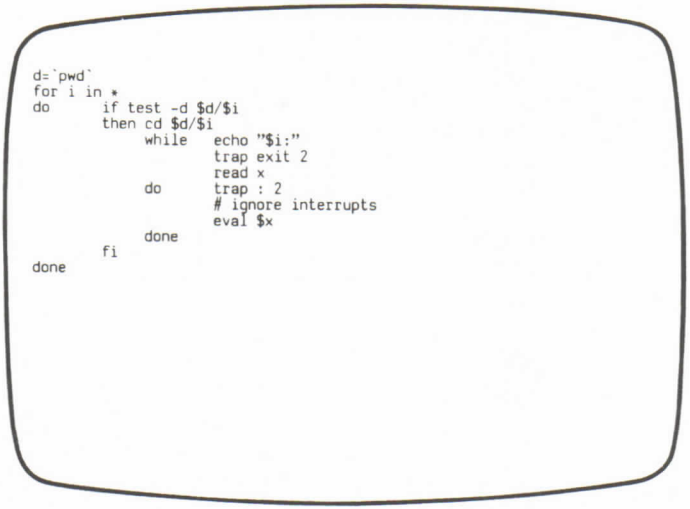
The **trap** command is most frequently used to make sure that temporary files are removed upon termination of a procedure. The preceding example would be written more typically as follows:



```
temp=$HOME/temp/$$
trap 'rm $temp; trap 0; exit' 0 1 2 3 15
ls > $temp
     # commands that use $temp here
```

In this example, whenever signal 1 (hangup), 2 (interrupt), 3 (quit), or 15(terminate) is received by the shell procedure, or whenever the shell procedure is about to exit, the commands enclosed between the single quotation marks are executed. The **exit** command must be included, or else the shell continues reading commands where it left off when the signal was received. The "trap " in the above procedure turns off the original traps 1, 2, 3, and 15 on exits from the shell, so that the **exit** command does not reactivate the execution of the **trap** commands.

Sometimes the shell continues reading commands after executing **trap** commands. The following procedure takes each directory in the current directory, changes to that directory, prompts with its name, and executes commands typed at the terminal until an end-of-file (**CTRL D**) or an interrupt is received. An end-of-file causes the **read** command to return a nonzero exit status, and thus the **while** loop terminates and the next directory cycle is initiated. An interrupt is ignored while executing the requested commands, but causes termination of the procedure when it is waiting for input:



```
d=`pwd`
for i in *
do      if test -d $d/$i
        then cd $d/$i
             while   echo "$i:"
                     trap exit 2
                     read x
             do      trap : 2
                     # ignore interrupts
                     eval $x
             done
        fi
done
```

Several traps may be in effect at the same time: if multiple signals are received simultaneously, they are serviced in numerically ascending order. To determine which traps are currently set, type:

```
trap
```

It is important to understand the way in which the shell implements the **trap** command. When a signal (other than 11) is received by the shell, it is passed on to whatever child processes are currently executing. When these (synchronous) processes terminate, normally or abnormally, the shell polls any traps that happen to be set and executes the appropriate **trap** commands. This process is straightforward, except in the case of traps set at the command (outermost, or login) level. In this case, it is possible that no child process is running, so before the shell polls the traps, it waits for the termination of the first process spawned after the signal was received.

When a signal is redefined in a shell script, this does not redefine the signal for programs invoked by that script; the signal is merely passed along. A disabled signal is not passed.

For internal commands, the shell normally polls traps on completion of the command. An exception to this rule is made for the **read** command, for which traps are serviced immediately, so that **read** can be interrupted while waiting for input.

SPECIAL SHELL COMMANDS

Several special commands are internal to the shell. The shell does not fork to execute these commands, so no additional processes are spawned. These commands should be used whenever possible, because they are, in general, faster and more efficient than other XENIX commands. The trade-off for this efficiency is that redirection of input and output is not allowed for most of these special commands.

Several of the special commands have already been described because they affect the flow of control. They are **dot** (.), **break, continue, exit,** and **trap.** The **set** command is also a special command. The remaining special commands are:

: Does nothing (the **null** command). Can be used to insert comments in shell procedures. Its exit status is zero (true). Its utility as a comment character has largely been supplanted by the number sign (#) which can insert comments to the end-of-line. Beware: any arguments to the **null** command are parsed for syntactic correctness; when in doubt, quote such arguments. Parameter substitution takes place, just as in other commands.

cd *arg* Makes the current directory. If *arg* is not a valid directory, or the user is not authorized to access it, a nonzero exit status is returned. Specifying **cd** *arg* with no *arg* is equivalent to typing "*cd* $HOME" which takes you to your home directory.

exec *arg...* Executes the command (if *arg* is a command) without forking and returning to the current shell. This is effectively a "goto" and no new process is created. Input and output redirection arguments are allowed on the command line. If only input and output redirection arguments appear, then the input and output of the shell itself are modified accordingly.

newgrp *arg...* Executes the **newgrp** command, replacing the shell. **newgrp** in turn creates a new shell. Beware: only environment variables will be known in the shell created by the **newgrp** command. Any exported variables will no longer be marked as such.

read *var...* Reads one line (up to a newline) from the standard input and assigns the first word to the first variable, the second word to the second variable, and so on. All words left over are assigned to the last variable. The exit status of read is zero unless an end-of-file is read.

readonly *var...* Makes the specified variables readonly so that no subsequent assignments may be made to them. If no arguments are given, a list of all readonly and of all exported variables is given.

times Prints the accumulated user and system times for processes run from the current shell.

umask *nnn* Sets the user file creation mask to *nnn* . If *nnn* is omitted, the current value of the mask is printed. This bit-mask is used to set the default permissions when creating files. For example, an octal umask of 137 corresponds to the following bit-mask and permission settings for a newly created file:

User	User	Group	Other
Octal	1	3	7
bit-mask	001	011	111
permissions	rw-	r--	---

See **umask(C)** in the XENIX User and System Administrator Reference Manual for information on the value of *nnn* .

wait Makes the shell wait for all currently active child processes to terminate. The exit status of wait is always zero.

CREATION AND ORGANIZATION OF SHELL PROCEDURES

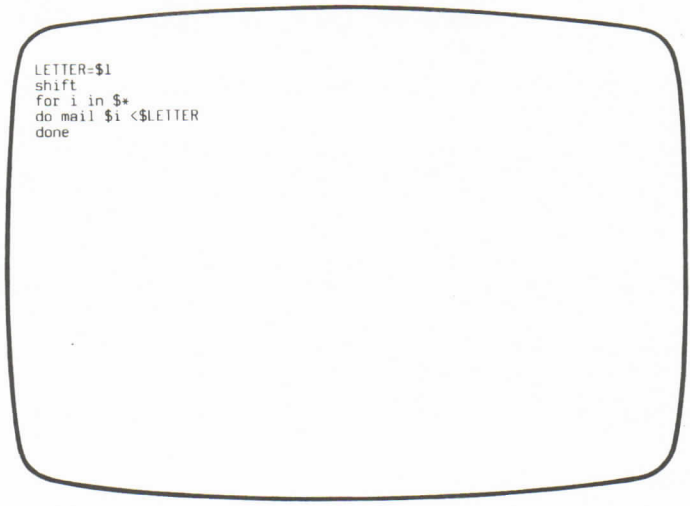
A shell procedure can be created in two simple steps. The first is building an ordinary text file. The second is changing the mode of the file to make it executable, thus permitting it to be invoked by:

```
proc args
```

rather than:

```
sh proc args
```

The second step may be omitted if a procedure is to be used once or twice and then discarded, but is recommended for frequently-used ones. To set up a simple procedure, first create a file named mailall with the following contents:



```
LETTER=$1
shift
for i in $*
do mail $i <$LETTER
done
```

Next type:

```
chmod +x mailall
```

The new command can then be invoked from within the current directory by typing:

```
mailall letter joe bob
```

letter is the name of the file containing the message you want to send, and joe and bob are people you want to send the message to. Note that shell procedures must always be at least readable, so that the shell itself can read commands from the file.

If mailall were thus created in a directory whose name appears in the user's PATH variable, the user could change working directories and still invoke the **mailall** command.

Shell procedures may be created dynamically. A procedure may generate a file of commands, and then invoke another instance of the shell to execute and remove the file. Or the **dot** command (.) can make the current shell read commands from the new file, which allows use of existing shell variables and avoids the spawning of an additional process for another shell.

Many users prefer writing shell procedures to writing C programs. This is true for several reasons:

1. A shell procedure is easy to create and maintain because it is only a file of ordinary text.

2. A shell procedure has no corresponding object program that must be generated and maintained.

3. A shell procedure is easy to create quickly, use a few times, and then remove.

4. Because shell procedures are usually short in length, written in a high-level programming language, and kept only in their source-language form, they are generally easy to find, understand, and modify.

By convention, directories that contain only commands and shell procedures are named bin. This name is derived from the word "binary", and is used because compiled and executable programs are often called "binaries" to distinguish them from program source files. Most groups of users sharing common interests have one or more bin directories set up to hold common procedures. Some users have their PATH variable list several such directories, although an excess of them may cause a decrease in system efficiency.

MORE ABOUT EXECUTION FLAGS

Several execution flags available in the shell can be useful in shell procedures:

-e This flag causes the shell to exit immediately if any command that it executes exits with a nonzero exit status. This flag is useful for shell procedures composed of simple command lines; it is not intended for use in conjunction with other conditional constructs.

-u This flag causes unset variables to be considered errors when substituting variable values. This flag can be used to effect a global check on variables, rather than using conditional substitution to check each variable.

-t This flag causes the shell to exit after reading and executing the commands on the remainder of the current input line. This flag is typically used by C programs which call the shell to execute a single command.

-n This is a "don't execute" flag. On occasion, you may want to check a procedure for syntax errors without executing the commands in the procedure. Using "set -nv" at the beginning of a file will accomplish this.

-k This flag causes all arguments of the form variable=value to be treated as keyword parameters. When this flag is not set, only such arguments that appear before the command name are treated as keyword parameters.

SUPPORTING COMMANDS AND FEATURES

Shell procedures can make use of any XENIX command. The commands described in this section are used frequently in shell procedures, or have been explicitly designed for such use.

CONDITIONAL EVALUATION: test

The **test** command evaluates the expression specified by its arguments and returns a zero exit status if the expression is true. If the expression is false, test returns a nonzero (false) exit status. Test also returns a nonzero exit status if it has no arguments. Often it is convenient to use the **test** command as the first command in the command list following an **if** or a **while**. Shell variables used in test expressions should be enclosed in double quotation marks if there is any chance of their being null or not set.

The square brackets may be used as an alias to test, so that:

[*expression*]

has the same effect as:

test *expression*

Note that the spaces before and after the expression in brackets are essential.

The following is a partial list of the options that can be used to construct a conditional expression:

-r *file*	True if the named file exists and is readable by the user.
-w *file*	True if the named file exists and is writable by the user.
-x *file*	True if the named file exists and is executable by the user.
-s *file*	True if the named file exists and has a size greater than zero.
-d *file*	True if the named file is a directory.
-f *file*	True if the named file is an ordinary file.
-z *s1*	True if the length of string *s1* is zero.
-n *s1*	True if the length of the string *s1* is nonzero.
-t *fildes*	True if the open file whose file descriptor number is *fildes* is associated with a terminal device. If *fildes* is not specified, file descriptor 1 is used by

default.

s1 = *s2* — True if strings *s1* and *s2* are identical.

s1 != *s2* — True if strings *s1* and *s2* are not identical.

s1 — True if *s1* is not the null string.

n1 -eq *n2* — True if the integers *n1* and *n2* are algebraically equal; other algebraic comparisons are indicated by **-ne** (not equal), **-gt** (greater than), **-ge** (greater than or equal to), **-lt** (less than) and **-le** (less than or equal to).

The options can be combined with the following operators:

! — Unary negation operator.

-a — Binary logical AND operator.

-o — Binary logical OR operator; it has lower precedence than the logical AND operator (**-a**).

(*expr*) — Parentheses for grouping; they must be escaped to remove their significance to the shell. In the absence of parentheses, evaluation proceeds from left to right.

Note that all options, operators, filenames, etc. are separate arguments to test.

ECHOING ARGUMENTS

The **echo** command has the following syntax:

echo [*options*] [*args*]

echo copies its arguments to the standard output. Each is followed by a single space, except for the last argument, which is normally followed by a newline. **echo** is often used to prompt the user for input, to issue diagnostics in shell procedures, or to add a few lines to an output stream in the middle of a pipeline. Another use is to verify the argument list generation process before issuing a command that does something drastic. The command:

```
ls
```

is often replaced by:

```
echo *
```

because the latter is faster and prints fewer lines of output.

The -n option to echo removes the newline from the end of the echoed line. Thus, the following two commands prompt for input and then allow typing on the same line as the prompt:

```
echo -n 'enter name:'
read name
```

The **echo** command also recognizes several escape sequences described in **echo**(C) in the XENIX User and System Administrator Reference Manual.

EXPRESSION EVALUATION: expr

The **expr** command provides arithmetic and logical operations on integers and some pattern-matching facilities on its arguments. It evaluates a single expression and writes the result on the standard output. **expr** can be used inside grave accents to set a variable. Some typical examples follow:

```
#       increment $A
A=`expr $a + 1`
#       put third through last characters of
#       $1 into substring
substring=`expr "$1" : '.. (.* )' `
#       obtain length of $1
c=`expr "$1" :  '.* ' `
```

The most common uses of **expr** are in counting iterations of a loop and in using its pattern-matching capability to pick apart strings.

TRUE AND FALSE

The **true** and **false** commands perform the functions of exiting with zero and nonzero exit status, respectively. The **true** and **false** commands are often used to implement unconditional loops. For example, you might type:

```
while true
     do echo forever
     done
```

This will echo forever on the screen until an **INTERRUPT** is typed.

IN-LINE INPUT DOCUMENTS

Upon seeing a command line of the form:

command << *eofstring*

where *eofstring* is any arbitrary string, the shell will take the subsequent lines as the standard input of *command* until a line is read consisting only of *eofstring* . (By appending a minus [-] to the input redirection symbol [<<], leading spaces and tabs are deleted from each line of the input document before the shell passes the line to *command* .)

The shell creates a temporary file containing the input document and performs variable and command substitution on its contents before passing it to the command.

Pattern matching on filenames is performed on the arguments of command lines in command substitutions. In order to prohibit all substitutions, you may quote any character of *eofstring* :

command <<\ *eofstring*

The in-line input document feature is especially useful for small amounts of input data, where it is more convenient to place the data in the shell procedure than to keep it in a separate file. For instance, you could type:

```
cat <<- xx
     This message will be printed on the
     terminal with leading tabs and spaces
     removed.
xx
```

This in-line input document feature is most useful in shell procedures. Note that in-line input documents may not appear within grave accents.

INPUT/OUTPUT REDIRECTION USING FILE DESCRIPTORS

We mentioned above that a command occasionally directs output to some file associated with a file descriptor other than 1 or 2. In languages such as C, one can associate output with any file descriptor by using the **write**(S) system call (see the XENIX User and System Administrator Reference Manual). The shell provides its own mechanism for creating an output file associated with a particular file descriptor. By typing:

fd1 >& *fd2*

where *fd1* and *fd2* are valid file descriptors, one can direct output that would normally be associated with file descriptor *fd1* to the file associated with *fd2* . The default value for *fd1* and *fd2* is 1. If, at run time, no file is associated with *fd2* , then the redirection is void.

The most common use of this mechanism is that of directing standard error output to the same file as standard output. This is accomplished by typing:

command 2>&1

If you wanted to redirect both standard output and standard error output to the same file, you would type:

command 1> *file* 2>&1

The order here is significant: first, file descriptor 1 is associated with *file* ; then file descriptor 2 is associated with the same file as is currently associated with file descriptor 1. If the order of the redirections were reversed, standard error output would go to the terminal, and standard output would go to *file* , because at the time of the error output redirection, file descriptor 1 still would have been associated with the terminal.

This mechanism can also be generalized to redirect standard input. You could type:

fda <& *fdb*

to cause both file descriptors *fda* and *fdb* to be associated with the same input file. If *fda* or *fdb* is not specified, file descriptor 0 is assumed. Such input redirection is useful for a command that uses two or more input sources.

CONDITIONAL SUBSTITUTION

Normally, the shell replaces occurrences of **$** *variable* by the string value assigned to *variable* , if any. However, a special notation allows conditional substitution, depending on whether the variable is set or not null. By definition, a variable is set if it has ever been assigned a value. The value of a variable can be the null string, which may be assigned to a variable in anyone of the following ways:

```
A=
bcd=""
efg=''
set '' ""
```

The first three examples assign null to each of the corresponding shell variables. The last example sets the first and second positional parameters to null. The following conditional expressions depend on whether a variable is set and not null. Note that the meaning of braces in these expressions differs from their meaning when used in grouping shell commands, and that here, parameter refers to either a digit or a variable name.

${** *variable* **:-** *string* **}** — If *variable* then substitute the value **$ *variable* in place of this expression. Otherwise, replace the expression with *string* . Note that the value of *variable* is not changed by the evaluation of this expression.

${** *variable* **:=** *string* **}** — If *variable* is set and is nonnull, then substitute the value **$ *variable* in place of this expression. Otherwise, set *variable* *string* , and then substitute the value **$** *variable* in place of this expression. Positional parameters may not be assigned values in this fashion.

${ *variable* **:?** *string* **}** — If *variable* is set and is nonnull, then substitute the value of *variable* for the *expression* . Otherwise, print a message of the form:

variable : *string*

and exit from the current shell. (If the shell is the login shell, it is not exited.) If *string* is omitted in this form, then the message:

variable : **parameter null or not set**

is printed instead.

${ *variable* **:+** *string* **}** — If *variable* is set and is nonnull, then substitute *string* for this expression.

Otherwise, substitute the null string. Note that the value of *variable* is not altered by the evaluation of this expression.

These expressions may also be used without the colon. In this variation, the shell does not check whether the variable is null or not; it only checks whether the variable has ever been set.

The following examples illustrate the use of this facility:

1. This example performs an explicit assignment to the PATH variable:

   ```
   "PATH"=${PATH:-':/bin:/usr/bin'}
   ```

   This says, if PATH has ever been set and is not null, then it keeps its current value; otherwise, set it to the string ":/bin:/usr/bin".

2. This example automatically assigns the **HOME** variable a value:

   ```
   cd ${HOME:='/usr/gas'}
   ```

   If **HOME** is set, and is not null, then change directory to it. Otherwise set **HOME** to the given value and change directory to it.

INVOCATION FLAGS

Four flags can be specified on the shell command line. These flags may not be turned on with the **set** command:

-i If this flag is specified, or if the shell's input and output are both attached to a terminal, the shell is interactive. In such a shell, INTERRUPT (signal 2) is caught and ignored, and TERMINATE (signal 15) and QUIT (signal 3) are ignored.

-s If this flag is specified or if no input/output redirection arguments are given, the shell reads commands from standard input. Shell output is written to file descriptor 2. The shell you get upon logging into the system has the **-s** flag turned on.

-c When this flag is turned on, the shell reads commands from the first string following the flag. Remaining arguments are ignored. Use double quotation marks to enclose a multiword string in order to allow for variable substitution.

EFFECTIVE AND EFFICIENT SHELL PROGRAMMING

This section outlines strategies for writing efficient shell procedures, ones that do not waste resources in accomplishing their purposes. The primary reason for choosing a shell procedure to perform a specific function is to achieve a desired result at a minimum human cost. Emphasis should always be placed on simplicity, clarity, and readability, but efficiency can also be gained through awareness of a few design strategies. In many cases, an effective redesign of an existing procedure improves its efficiency by reducing its size, and often increases its comprehensibility. In any case, you needn't plan to optimize shell procedures unless they are intolerably slow or are known to consume an inordinate amount of system resources.

Shell procedures should undergo the same iteration cycle as other programs: write code, measure it, and optimize only the few important parts. The user should become familiar with the **time** command, which can be used to measure both entire procedures and parts thereof. Its use is strongly recommended; human intuition is notoriously unreliable when used to estimate timings of programs, even when the style of programming is a familiar one. Each timing test should be run several times, because the results are easily disturbed by variations in system load.

NUMBER OF PROCESSES GENERATED

When large numbers of short commands are executed, the actual execution time of the commands may well be dominated by the overhead of creating processes. The procedures that incur significant amounts of such overhead are those that perform much looping, and those that generate command sequences to be interpreted by another shell.

If you are worried about efficiency, it is important to know which commands are currently built into the shell, and which are not. Here is an alphabetical list of built-in commands:

break	**exec**	**newgrp**	**test**	**wait**
case	**exit**	**read**	**times**	**while**
cd	**export**	**readonly**	**trap**	**.**
continue	**for**	**set**	**umask**	**:**
eval	**if**	**shift**	**until**	**{}**

Parentheses, (), are built into the shell, but commands enclosed within them are executed as a child process, i.e., the shell does a **fork** , but no **exec** . Any command not in the above list requires both **fork** and **exec**.

You should always have at least a vague idea of the number of processes generated by a shell procedure. In the bulk of observed procedures, the number of processes created (not necessarily simultaneously) can be described by:

$$processes = (k * n) + c$$

where k and c are constants, and n may be the number of procedure

arguments, the number of lines in some input file, the number of entries in some directory, or some other obvious quantity. Efficiency improvements are most commonly gained by reducing the value of *k* , sometimes to zero. Any procedure whose complexity measure includes *n2* terms or higher powers of *n* is likely to be intolerably expensive.

As an example, here is an analysis of a procedure named split, whose text is given below:

```
#       split
trap 'rm temp$$; trap 0; exit' 0 1 2 3 15
start1=0 start2=0
b='[A-Za-z]'
cat > temp$$
        # read stdin into temp file
        # save original lengths of $1, $2
if test -s "$1"
then start1=`wc -l < $1`
fi
if test -s "$2"
then start2=`wc -l < $2`
fi
grep "$b" temp$$ >> $1
        # lines with letters onto $1
grep -v "$b" temp$$ | grep '[0-9]' >> $2
        # lines with only numbers onto $2
total="`wc -l < temp$$`"
end1="`wc -l < $1`"
end2="`wc -l < $2`"
lost="`expr $total -  ($end1 - $start1 )
-  ($end2 - $start2 )`"
echo "$total read, $lost thrown away"
```

For each iteration of the loop, there is one **expr** plus either an **echo** or another **expr.** One additional **echo** is executed at the end. If *n* is the number of lines of input, the number of processes is 2* *n* +1.

Some types of procedures should not be written using the shell. For example, if one or more processes are generated for each character in a file, it is a good indication that the procedure should be rewritten in C. Shell procedures should not be used to scan or build files a character at a time.

NUMBER OF DATA BYTES ACCESSED

It is worthwhile to consider any action that reduces the number of bytes read or written. This may be important for those procedures whose time is spent passing data around among a few processes, rather than in creating large numbers of short processes. Some filters shrink their output, others usually increase it. It always pays to put the shrinkers first when the order is irrelevant. For instance, the second of the following examples is likely to be faster since the input to sort will be much smaller:

```
sort file | grep pattern
grep pattern file | sort
```

SHORTENING DIRECTORY SEARCHES

Directory searching can consume a great deal of time, especially in those applications that utilize deep directory structures and long pathnames. Judicious use of **cd**, the change directory command, can help shorten long pathnames and thus reduce the number of directory searches needed. As an exercise, try the following commands:

```
ls -l /usr/bin/* >/dev/null
cd /usr/bin; ls -l * >/dev/null
```

The second command will run faster because of the fewer directory searches.

DIRECTORY SEARCH ORDER AND THE PATH VARIABLE

The PATH variable is a convenient mechanism for allowing organization and sharing of procedures. However, it must be used in a sensible fashion, or the result may be a great increase in system overhead.

The process of finding a command involves reading every directory included in every pathname that precedes the needed pathname in the current PATH variable. As an example, consider the effect of invoking **nroff** (i.e., /usr/bin/nroff) when the value of PATH is ":/bin:/usr/bin". The sequence of directories read is:

```
.
/
/bin
/
/usr
/usr/bin
```

This is a total of six directories. A long path list assigned to PATH can increase this number significantly.

The vast majority of command executions are of commands found in /bin and, to a somewhat lesser extent, in /usr/bin. Careless PATH setup may lead to a great deal of unnecessary searching. The following four examples are ordered from worst to best with respect to the efficiency of command searches:

```
:/usr/john/bin:/usr/localbin:/bin:/usr/bin
:/bin:/usr/john/bin:/usr/localbin:/usr/bin
:/bin:/usr/bin:/usr/john/bin:/usr/localbin
/bin::/usr/bin:/usr/john/bin:/usr/localbin
```

The first one above should be avoided. The others are acceptable and the choice among them is dictated by the rate of change in the set of commands kept in /bin and /usr/bin.

A procedure that is expensive because it invokes many short-lived commands may often be speeded up by setting the PATH variable inside the procedure so that the fewest possible directories are searched in an optimum order.

GOOD WAYS TO SET UP DIRECTORIES

It is wise to avoid directories that are larger than necessary. You should be aware of several special sizes. A directory that contains entries for up to 30 files (plus the required . and ..) fits in a single disk block and can be searched very efficiently. One that has up to 286 entries is still a small directory; anything larger is usually a disaster when used as a working directory. It is especially important to keep login directories small, preferably one block at most. Note that, as a rule, directories never shrink. This is very important to understand, because if your directory ever exceeds either the 30 or 286 thresholds, searches will be inefficient; furthermore, even if you delete files so that the number of files is less than either threshold, the system will still continue to treat the directory inefficiently.

SHELL PROCEDURE EXAMPLES

The power of the XENIX shell command language is most readily seen by examining how XENIX's many labor-saving utilities can be combined to perform powerful and useful commands with very little programming effort. This section gives examples of procedures that do just that. By studying these examples, you will gain insight into the techniques and shortcuts that can be used in programming shell procedures (also called "scripts"). Note the use of the number sign (#) to introduce comments into shell procedures.

Carry out the following steps for each procedure:

1. Place the procedure in a file with the indicated name.

2. Give the file execute permission with the **chmod** command.

3. Move the file to a directory in which commands are kept, such as your own bin directory.

4. Make sure that the path of the bin directory is specified in the PATH variable found in .profile.

5. Execute the named command.

BINUNIQ

```
ls /bin /usr/bin | sort | uniq -d
```

This procedure determines which files are in both /bin and /usr/bin. It is done because files in /bin will override those in /usr/bin during most searches, and duplicates need to be weeded out. If the /usr/bin file is obsolete, then space is being wasted; if the /bin file is outdated by a corresponding entry in /usr/bin, then the wrong version is being run and, again, space is being wasted. This is also a good demonstration of "sort | uniq" to find matches and duplications.

COPYPAIRS

```
#       Usage: copypairs file1 file2 ...
#       Copies file1 to file2, file3 to file4, ...
while test "$2" != ""
do
     cp $1 $2
     shift; shift
done
if test "$1" != ""
     then echo "$0: odd number of arguments"
fi
```

This procedure illustrates the use of a while loop to process a list of positional parameters that are somehow related to one another. Here a **while** loop is much better than a **for** loop, because you can adjust the positional parameters with the **shift** command to handle related arguments.

COPYTO

```
#       Usage: copyto dir file ...
#       Copies argument files to "dir",
#       making sure that at least
#       two arguments exist, that "dir" is a directory,
#       and that each additional argument
#       is a readable file.
if test $# -lt 2
     then echo "$0: usage: copyto directory file ..."
elif test ! -d $1
     then echo "$0: $1 is not a directory";
else    dir=$1; shift
     for eachfile
     do      cp $eachfile $dir
done
fi
```

This procedure uses an **if** command with several parts to screen out improper usage. The **for** loop at the end of the procedure loops over all of the arguments to **copyto** but the first; the original **$1** is shifted off.

DISTINCT1

```
#       Usage: distinct1
#       Reads standard input and reports list of
#       alphanumeric strings that differ only in case,
#       giving lowercase form of each.
tr -cs 'A-Za-z0-9' ' 012'|sort -u |
tr 'A-Z' 'a-z' | sort | uniq -d
```

This procedure demonstrates the kind of process that is created by the left-to-right construction of a long pipeline. Note the use of the backslash at the end of the first line as the line continuation character. It may not be immediately obvious how this command works. You may wish to consult **tr(C)**, **sort(C)**, and **uniq(C)** in the XENIX Reference Manual if you are completely unfamiliar with these commands. The **tr** command translates all characters except letters and digits into newline characters, and then squeezes out repeated newline characters. This leaves each string (in this case, any contiguous sequence of letters and digits) on a separate line. The **sort** command sorts the lines and emits only one line from any sequence of one or more repeated lines. The next **tr** converts everything to lowercase, so that identifiers differing only in case become identical. The output is sorted again to bring such duplicates together. The "uniq -d" prints (once) only those lines that occur more than once, yielding the desired list.

The process of building such a pipeline relies on the fact that pipes and files can usually be interchanged. In the following example, the first line is equivalent to the last two lines, assuming that sufficient disk space is available:

```
cmd1 | cmd2 | cmd3

cmd1 > temp1; < temp1 cmd2 > temp2; < temp2 cmd3
rm temp[123]
```

Starting with a file of test data on the standard input and working from left to right, each command is executed taking its input from the previous file and putting its output in the next file. The final output is then examined to make sure that it contains the expected result. The goal is to create a series of transformations that will convert the input to the desired output.

Although pipelines can give a concise notation for complex processes, you should exercise some restraint, since such practice often yields incomprehensible code.

DRAFT

```
#       Usage: draft file(s)
#       Print manual pages for Diablo printer.
for i in $*
    do nroff -man $i | lpr
done
```

Users often write this kind of procedure for convenience in dealing with commands that require the use of distinct flags that cannot be given default values that are reasonable for all (or even most) users.

EDFIND

```
#    Usage: edfind file arg
#    Finds the last occurrence in "file" of a line
#    whose beginning matches "arg", then prints
#    3 lines (the one before, the line itself,
#    and the one after)
ed - $1 <<-EOF
     ?^$2?
     -,+p
     q
EOF
```

This illustrates the practice of using **ed** in-line input scripts into which the shell can substitute the values of variables.

EDLAST

```
#       Usage: edlast file
#       Prints the last line of file,
#       then deletes that line.
ed - $1 <<- !
       $p
       $d
       w
       q
!
echo done
```

This procedure illustrates taking input from within the file itself up to the exclamation point (!). Variable substitution is prohibited within the input text because of the backslash.

FSPLIT

```
#     Usage: fsplit file1 file2
#     Reads standard input and divides it into 3 parts
#     by appending any line containing at least one letter
#     to file1, appending any line containing digits but
#     no letters to file2, and by throwing the rest away.
count=0 gone=0
while read next
do
      count="`expr $count + 1`"
      case "$next" in
      *[A-Za-z]*)
            echo "$next" >> $1 ;;
      *[0-9]*)
            echo "$next" >> $2 ;;
      *)
            gone="`expr $gone + 1`"
      esac
done
echo "$count lines read, $gone thrown away"
```

Each iteration of the loop reads a line from the input and analyzes it. The loop terminates only when read encounters an end-of-file. Note the use of the **expr** command.

Don't use the shell to read a line at a time unless you must -- it can be an extremely slow process.

LISTFIELDS

```
grep $* | tr ":" "\012"
```

This procedure lists lines containing any desired entry that is given to it as an argument. It places any field that begins with a colon on a newline. Thus, if given the following input:

```
joe newman: 13509 NE 78th St: Redmond, Wa 98062
```

listfields will produce this:

```
joe newman
13509 NE 78th St
Redmond, Wa 98062
```

Note the use of the **tr** command to transpose colons to linefeeds.

```
#     Usage: mkfiles pref [quantity]
#     Makes "quantity" files, named pref1, pref2, ...
#     Default is 5 as determined on following line.
quantity=${2-5}
i=1
while test "$i" -le "$quantity"
do
      > $1$i
      i="`expr $i + 1`"
done
```

The **mkfiles** procedure uses output redirection to create zero-length files. The **expr** command is used for counting iterations of the **while** loop.

NULL

```
#    Usage: null files
#    Create each of the named files as an empty file.
for each file
do
     >$eachfile
done
```

This procedure uses the fact that output redirection creates the (empty) output file if a file does not already exist.

PHONE

```
#    Usage: phone initials ...
#    Prints the phone numbers of the
#    people with the given initials.
echo 'inits  ext      home'
grep "^$1" <<-END
     jfk      1234     999-2345
     lbj      2234     583-2245
     hst      3342     988-1010
     jqa      4567     555-1234
END
```

This procedure is an example of using an in-line input script to maintain

a small data base.

TEXTFILE

```
if test "$1" = "-s"
then
#     Return condition code
      shift
      if test -z "`$0 $*`" # check return value
      then
              exit 1
      else
              exit 0
      fi
fi

if test $# -lt 1
then      echo "$0: Usage: $0 [ -s ] file ..." 1>&2
       exit 0
fi

file $* |  fgrep ' text' | sed 's:   .*//'
```

To determine which files in a directory contain only textual information, textfile filters argument lists to other commands. For example, the following command line will print all the text files in the current directory:

```
pr `textfile *` | lpr
```

This procedure also uses an -s flag which silently tests whether any of the files in the argument list is a text file.

WRITEMAIL

```
#       Usage: writemail message user
#       If user is logged in,
#       writes message to terminal;
#       otherwise, mails it to user.
echo "$1" | { write "$2" || mail "$2" ;}
```

This procedure illustrates the use of command grouping. The message specified by **$1** is piped to both the **write** command and, if write fails, to the **mail** command.

SHELL GRAMMAR

Feature	Syntax
item :	*word* *input-output* *name = value*
simple-command :	*item* *simple-command item*
command:	*simple-command* *(command-list)* *{ command-list }* *for name do command-list done* *for name in word do command-list done* *while command-list do command-list done* *until command-list do command-list done* *case word in case-part esac* *if command-list then command-list else-part fi*
pipeline :	*command* *pipeline | command*
andor :	*pipeline* *andor && pipeline*

```
                    andor || pipeline

command-list :      andor
                    command-list ;
                    command-list &
                    command-list ; andor
                    command-list & andor

input-output :      > file
                    < file
                    << word
                    >> word

file :              word
                    & digit
                    & -

case-part :         pattern ) command-list ;;

pattern :           word
                    pattern | word

else-part :         elif command-list then command-list else-part
                    else command-list


empty               empty

word :              a sequence of nonblank characters

name :              a sequence of letters, digits, or underscores
                    starting with a letter

digit :             0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

METACHARACTERS AND RESERVED WORDS

Syntax

```
|       Pipe symbol
&&      And-if symbol
||      Or-if symbol
;       Command separator
;;      Case delimiter
&       Background commands
()      Command grouping
<       Input redirection
<<      Input document from here
>       Output creation
>>      Output append
#       Comment to end of line
```

Patterns

*	Match any character(s) including none
?	Match any single character
[...]	Match any of the enclosed characters

Substitution

${...}	Substitute shell variable
`...`	Substitute command output

Quoting

\	Quote next character as literal with no special meaning
'...'	Quote enclosed characters excepting the single, open quotation marks (')
"..."	Quote enclosed characters excepting: $ ` \ "

Reserved Words

case	for
do	if
done	in
elif	then
else	until
esac	while
fi	{ }

8. THE VISUAL SHELL

ABOUT THIS CHAPTER

This chapter serves as an introduction to the XENIX V Visual Shell.

CONTENTS

GETTING STARTED

INTRODUCTION

The XENIX Visual Shell is an interface between you and your computer's operating system.

The user interface or "shell" is what you see on the screen when your system is waiting for a command. The shell interprets all commands to the operating system; it replaces the traditional command line with a visual shell that shows you a menu of the most commonly executed applications and utilities.

Pressing the **HELP** key or choosing the **Help** command displays **Help** information. Files for use with a command may be selected by pointing to a file or directory with the cursor. This saves you from having to type the whole filename or directory name, and reduces the possibility of selecting the wrong file or directory.

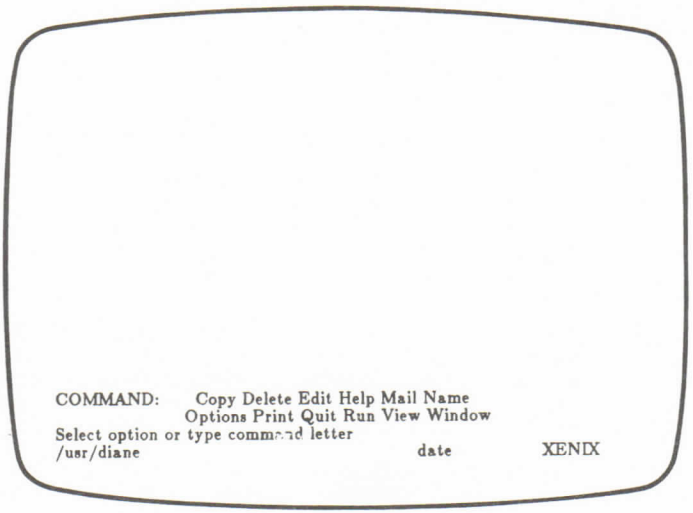
An important feature of the Visual Shell is that you can customize it for different purposes. You can create commands and help files as well as tailor the shell for different applications.

The Visual Shell typically will be used by people running prepackaged applications (such as Microsoft Multiplan) under XENIX. Users with more advanced knowledge will customize the menus for more efficient running of applications and data processing.

In this section, we will define some terms and introduce you to the Visual Shell environment.

COMMAND SCREEN

When you start XENIX the first thing you will see on the screen is the Visual Shell. This screen is called the command screen. Your screen may look something like this:



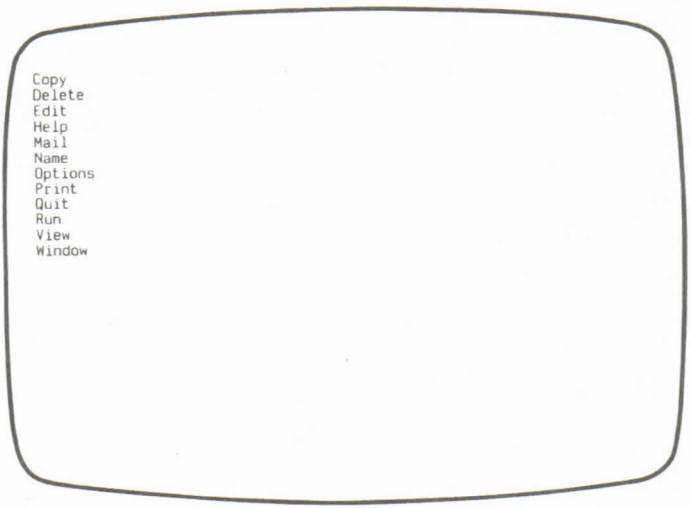
OUTPUT OF COMMANDS

The center portion of your screen appears blank when you start your computer. As you type commands, the output of these commands appears in the central portion of the screen. The command output scrolls up each time a new command is typed, until the directory window is reached. The output scrolls past the directory window as new output is displayed at the bottom. The command menu, message line, and status line are temporarily erased during command execution and are redrawn when the command has completed.

Normally, output in this area reflects the Visual Shell command you choose. A special command lets you see exactly which XENIX command you are running. See "Options Output" in the section entitled "Command Directory", for more information.

MAIN COMMAND MENU

The main command menu shows 12 commands:



You will use these commands to copy and delete files, run applications, quit the session, obtain "Help" text, and access many of the functions of the operating system. Below the menu is the prompt, "Choose option or type command letter". You will learn how to choose options and type command letters in the section entitled "Using the Visual Shell".

STATUS LINE

The status line is the last line on the screen. It gives you the following information:

- The pathname of the working directory (left truncated if necessary).
- A message if you have mail.
- The date.
- The time.
- The name of the operating system (XENIX).

When the operating system is waiting for you to enter commands, two things are monitored and displayed on the status line: the time and the date. The time is updated every minute, and the date is updated daily.

MESSAGE LINE

The message line is above the status line. The Visual Shell displays various messages on this line when you choose commands and execute operating system tasks. Any error messages you may receive appear on this line.

DEFINITIONS

You should understand the following terms when using the Visual Shell:

File — A file is a collection of related information. All programs, text, and data on your disks reside in files. You create a file each time you enter and save data or text at your terminal. Files are also created when you write and name programs and save them on your computer.

Filename — A filename is a string of up to 14 characters. Characters such as period (.) and hyphen (-) are allowed so you may create filename extensions that are used to identify types of files. An example of a filename with an extension is *newfile.mnu* .

Directory — The names of files are kept in directories on your computer. These directories also contain information on the size of the files, their location on disk, and the dates that they were created or modified. The directory you are working in is called your "working" directory.

The contents of the working directory can be listed on the screen by pressing one of the direction keys.

Pathname — A pathname is a sequence of directory names followed by a simple filename, each separated from the previous one by a forward slash (/). Pathnames are used to uniquely identify files that may have the same name or that are not in your current working directory. An example of a pathname is */usr/joe/testfile* .

Special Characters — The asterisk (*) and the question mark (?) special characters can be used when specifying filenames. These special characters give you greater flexibility when using filenames in XENIX commands.

The question mark (?) in a filename indicates any character occupying that position. The asterisk (*) in a filename indicates any character occupying that position or any of the remaining positions in the filename.

Switches

Switches are options that control operating system commands. They appear on command menus as choices, as in the following example:

```
recursive: Yes (No)
```

The default choice is always enclosed in parentheses.

Filter

A filter is a command that reads your input, transforms it in some way, and then outputs it, usually to your terminal or to a file. In this way, the data is said to have been "filtered" by the program. Since filters can be put together in many different ways, a few filters can take the place of a large number of specific commands.

Piping

If you want to have the output of one program sent as input to another, you can "pipe" commands to the operating system. For example, you may occasionally need to have the output of one program sent as the input to another program. A typical case would be a program that produces output in columns. In addition, you may want to have the columns sorted.

Piping with the Visual Shell is accomplished by typing a bar (|) symbol in the "output:" field of a command menu. The Visual Shell will then display a **Filter** menu. You can choose the appropriate filter (**Sort** , **More** , **Get** , etc.) and your program will be "piped" through that filter. You can also pipe programs and commands through your own filters.

You can pipe (redirect) the output of a command or program by specifying an output filename or device name in the "output:" command field.

USING THIS CHAPTER

This chapter contains examples that you can follow while learning XENIX. To help you follow the examples, refer to the section entitled "Key Directory," for a list of key functions.

You can also use the **Help** command any time to find out which key to use to perform a function. See "Getting Help: The Help Command" in the section entitled "Using the Visual Shell", for more information.

CHOOSING COMMANDS

COMMAND MENUS

The lists of commands you see on your screen are called menus. In fact, any time you see choices on the bottom part of the screen, that is a menu. You can choose an option from a menu by pressing **SPACE** . As you press **SPACE** , watch the command menu. The highlight moves left to right, stopping at each command. Try pressing **SPACE** to move to each of the different options on the main command menu. Once you have selected a command, press the **CR** key and a new menu will be displayed on the screen. You must fill in the blank fields of this submenu to tell the Visual Shell what to do with the command you selected.

The **BKSP** key can be used to back up (move right to left) through the commands.

To get back to the main command menu at any time, press the **CANCEL** key. Refer to the section entitled "Key Directory", for the key or keys that perform the **Cancel** function.

Another way to select commands and other options is to type the first letter of the command or option while in the command menu. For example, if you want to copy a file, type **C** . You do not need to press **CR** ; the **Copy** command will automatically be processed.

If you press a key that does not work as a command, such as the letter J, the command screen will not change, but you will see the error message:

Not a valid option.

EDITING RESPONSES TO COMMANDS

The Visual Shell provides special editing keys to edit responses in the command fields. The following keys are listed by *function* only. Refer to "Key Directory". for the exact key or sequence of keys that corresponds to these functions.

The field containing a proposed response is highlighted just after a command is selected or after pressing the **TAB** key.

To replace the proposed response, type the replacement. The Visual Shell automatically deletes the proposed response as soon as you type the new one.

To delete the proposed response and leave the field empty, press the **DELETE** key. All text that is highlighted is deleted.

To append to the proposed response, press either the **CHARACTER RIGHT** or the **WORD RIGHT** key, then type the additional text.

Once the proposed response is altered, one character or word in the field is highlighted. This highlight is the edit cursor. The edit cursor may

be moved to designate where or what to edit.

Use the **CHARACTER LEFT** , **CHARACTER RIGHT** , **WORD LEFT** , and **WORD RIGHT** keys to move the edit cursor in the command fields. The **CHARACTER LEFT** and **CHARACTER RIGHT** keys move the edit cursor left or right one character. The **WORD LEFT** and **WORD RIGHT** keys move the edit cursor left or right, choosing words or the space or punctuation between words.

To insert new text, type the text. It will be inserted in front of the edit cursor.

To delete text, use the **BKSP** key to delete characters on the left side of the cursor. Use the **DELETE** key to delete what is highlighted by the cursor.

To replace text, delete the old text and type the new text.

CARRYING OUT A COMMAND: THE CR KEY

The Visual Shell does not carry out the command until you tell it to do so by pressing the **CR** key. As shown earlier, the **CR** key is also used after you move the highlight to a command or subcommand name with the **SPACE** or **BKSP** keys.

You can press the **CR** key whenever the responses in all the command fields are correct. When a command has been carried out, the command screen reappears and waits for a new command. Note that if you choose a command by typing its first letter, you do not have to press the **CR** key.

CANCELLING A COMMAND: THE CANCEL KEY

Before you press **CR** to carry out a command, you may press the **CANCEL** key to cancel the command. When you press the **CANCEL** key, the command screen will reappear and you can choose another command or quit the session. Refer to the section entitled "Key Directory" to locate the **CANCEL** key on your keyboard.

REDRAWING THE SCREEN: THE REDRAW KEY

You may clear your screen at any time during the computer session by pressing the **REDRAW** key. Consult the section entitled "Key Directory" to determine which key corresponds to the **Redraw** function.

USING THE VISUAL SHELL

INTRODUCTION

This section discusses how to perform operating system functions by using Visual Shell commands. The topics include:

- Creating files.
- Managing files.
- Managing directories.
- Managing diskettes.
- Sending and receiving mail.
- Running applications.
- Getting help.
- Quitting.

CREATING FILES

The Visual Shell includes a specific command to help you create and edit files. This command is called the **Edit** command.

Using an Editor: the Edit Command

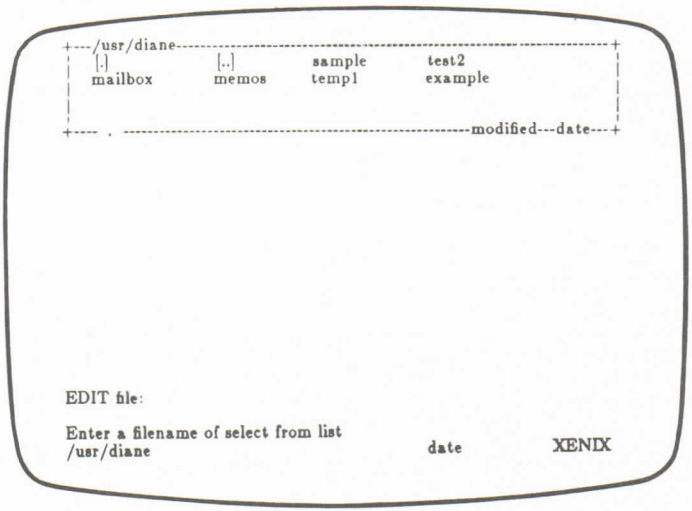
You can create and delete text and program files using the XENIX editor. The Visual Shell default editor for XENIX is **vi** . Refer to the section entitled "Using Advanced Features" for information on how to use other editors under XENIX.

You use the **Edit** command to automatically call up **vi**. When the **vi** menu appears, type the name of the file you want to create or edit, and then press the **CR** key. The Visual Shell will disappear and you will be using the editor. When you exit the editor, the Visual Shell is redisplayed.

Example:

This example creates a sample file named *test* in your working directory.

1. Choose the **Edit** command from the main command menu by typing the letter E. Your screen should look like this:



2. Type the filename *test* in the *filename:* command field.

3. Press the **CR** key.

   The Visual Shell will disappear, and you will be using **vi** . (For more information on **vi** , see the appropriate chapter of this guide.)

4. Type the letter I (for "Insert").

5. Type the following line:

   I'm a XENIX Visual Shell temporary file.

6. Press **ESC** .

7. To save this file in your working directory type ":x". We will be referring to this test file in later sections of this manual.

Viewing your Directory: the Direction Keys

You can scroll through your working directory anytime by pressing one of the direction keys: **LEFT** , **RIGHT** , **UP** , or **DOWN** . For example, to select the filename to the right of the current selection, press the **RIGHT** direction key.

MANAGING FILES

The commands in this section will help you manage your files. The commands discussed are: **View** , **Copy** , **Delete** , **Name** , **Options** and **Print**. These commands are used to view, copy, delete, rename, change permissions, and print files.

Viewing Files: the View Command

The **View** command is used to view files on the screen. You can scan through files using this command, but you cannot edit them. Use the **Edit** command to edit files.

To use the **View** command, choose the **View** command from the main command menu. Fill in the command menu with the name of the file you want to view. When you press the **CR** key, a five-line window will appear at the top of the screen. The file you selected will be displayed in that window. If the file is larger than the window, you can use the direction keys and some function keys to scroll the file in the window. Refer to the section entitled "Key Directory" for more information on function keys.

When the window appears on the screen, the main command menu will reappear. You can choose other Visual Shell commands while the window is on the screen.

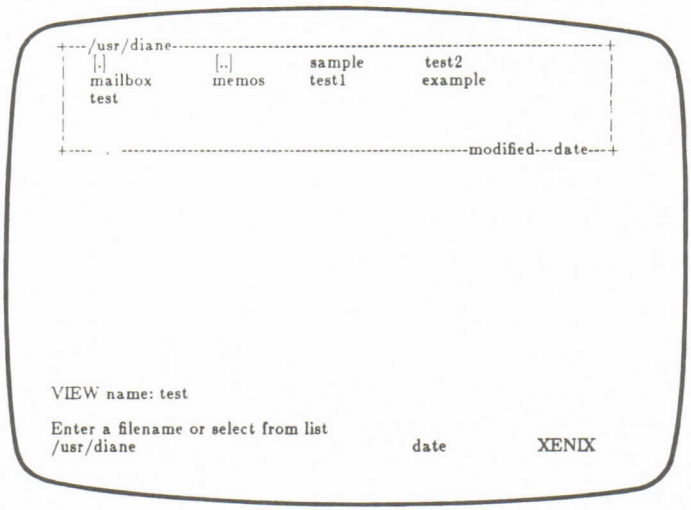
If you make a mistake, you can always press the **CANCEL** key to return to the main command menu. Refer to the section entitled "Key Directory" for information on the **CANCEL** key.

Example:

To view the *test* file that you created with the **Edit** command, follow these steps:

1. Choose the **View** command from the main command menu.

2. Fill in the menu by typing *test* in the *filename:* command field. Your screen should look like this:



3. Press the **CR** key.

   When you press the **CR** key, a window will open at the top of your screen. The *test* file will appear in this window.

Renaming Files: the Name Command

The **Name** command renames files. To use this command, choose the **Name** command from the main command menu. The **Name** menu will appear. Fill in the name of the file you want to rename in the "from:" command field, and the new name of the file in the "to:" command field. When you press **CR**, your file will be renamed. You will know that the renaming process is complete when you see a message in the command output area of the screen.

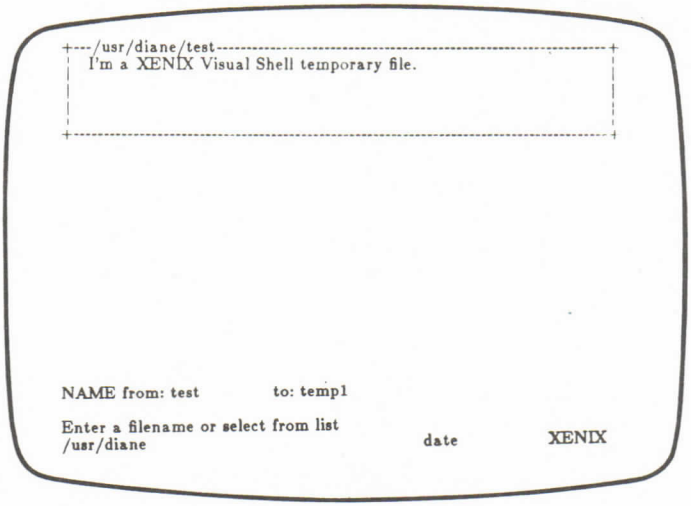
You can rename any file according to the rules for naming files. Refer to the first volume of this guide for more information on naming conventions.

Example:

To rename the *test* file *temp1* , follow these steps:

1. Choose the **Name** command from the main command menu.

2. Fill in the **Name** menu as follows:



3. Press **CR**

The *test* file is now renamed *temp1* . You will see the message, **Name** (1) **test** (2) **temp1** in the command qutput area of the screen.

Copying Files: the Copy File Command

The **Copy** command on the main command menu is used to copy files. It is also used to copy directories as described in later in this section.

To copy a file, choose the **Copy** command from the main command menu. The **Copy** menu will appear. This menu asks you whether you want to copy a file or a directory. When you choose **File** , the **Copy File** menu appears. You must type the name of the file you want to copy in the "from:" command field, and the new name of the file in the "to:" command field.

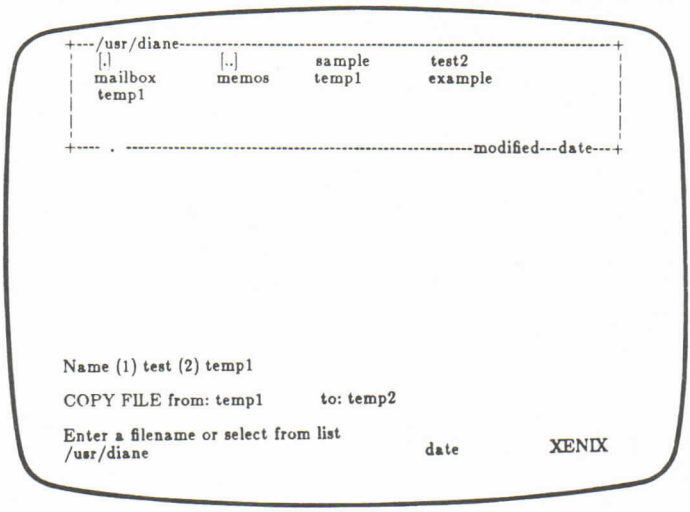
Example:

To make a copy of the *temp1* file and name the copy *temp2* , follow these steps:

1. Choose **Copy** from the main command menu.

2. Choose **File** from the **Copy** menu.

3. Fill in the **Copy File** menu command fields. Your screen should look like this:



4. Press the **CR** key.

The message, **Copy File (1) temp1 (2) temp2** will appear in the command output area of the screen when the *temp1* file has been copied. You now have two identical files in your working directory: *temp1* and *temp2* .

Deleting Files: the Delete Command

You can delete files and directories with the **Delete** command. For information on deleting directories, refer to the subsection entitled "Deleting Directories: the Delete Command"

To delete a file, choose **Delete** from the main command menu. Fill in the name of the file you wish to delete in the "name:" command field. When you press **CR** , your file will be deleted. You will see a message in the command output area of the screen.

Example:

To delete the file *temp2* from your working directory, follow these steps:

1. Choose the **Delete** command from the main command menu.

2. Type *temp2* in the "name:" command field.

3. Press **CR** .

The file *temp2* is deleted. You will see the message:

Delete (1) temp2

in the command output area of the screen.

Printing Files: the Print Command

The **Print** command is used to print files. To print a file, choose the **Print** command from the main command menu. The **Print** menu will appear. This menu contains a "file:" command field. To print a file or files, type the name of the file in the *filename:* command field. When you press **CR** , your file will be sent to the printer. The message:

Print (1) filename

will be displayed in the command output area.

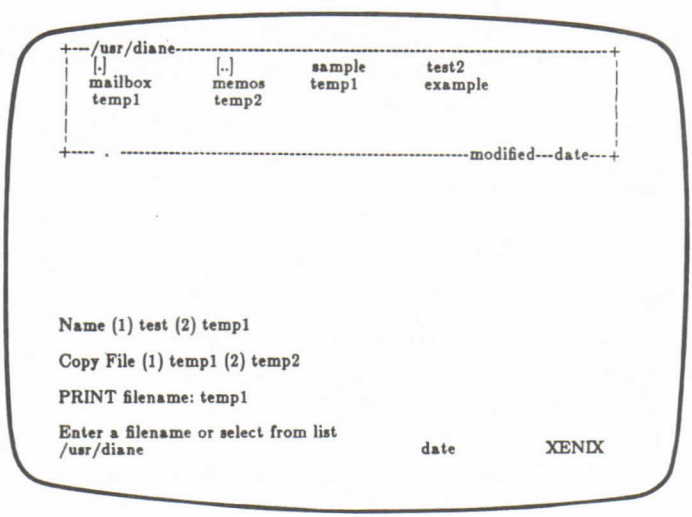
To print more than one file, separate the filenames with spaces when you fill in the *filename:* command field.

Make sure your printer is attached to your computer.

Example:

To print the file *temp1:*

1. Choose the **Print** command from the main command menu.
2. Type *temp1* in the *filename:* command field. Your screen should look like this:



3. Press the **CR** key.

The *templ* file is sent to the printer. When it has been printed, the Visual Shell will display the message:

```
Print (1) templ
```

in the command output area of the screen.

Setting File Permissions: the Options Permissions Command

The **Options** command is one of the Visual Shell commands that has a submenu of other commands. (The other Visual Shell commands with submenus are Copy and Mail.) Use the **Options** command to set file permissions to protect various files in your directory. You can set the file permission to a variety of combinations.

To set a file permission, choose the **Options** command from the main command menu. An **Options** menu will appear. Now, choose the **Permissions** command. Fill in the command fields with the name of the file you wish to protect and with the kind of protection desired. When you press **CR**, the permission will change on your file.

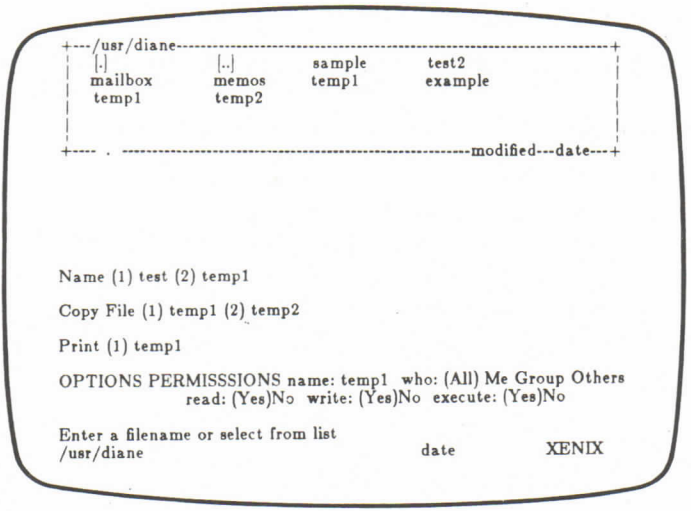
Example:

The following example sets read and write permission to "all users" for the file named *templ* .

1. Choose **Options** from the main command menu.

2. Choose **Permissions** from the **Options** menu.

3. Fill in the command fields so that your screen looks like this:



4. Press the **CR** key.

The file named *temp1* is now changed so that all persons on the system can read and write to the file.

To verify that the permissions on the file have changed, you must run the XENIX **ls -l** command. For more information refer to the section entitled "Listing Permissions: the Run Command" .

MANAGING DIRECTORIES

The commands in this section will help you manage your directories. These commands are used to create and delete directories, transfer to a new working directory, change permissions on directories, and determine disk usage.

Creating a Directory: the Options Directory Make Command

The **Options** menu contains several commands that perform directory functions. The **Options Directory Make** command creates new directories on your system. To create a directory, choose the **Options** command from the main menu. An **Options** menu will appear. Select the **Directory** command. The **Options Directory** menu will appear. When you choose the **Make** command, you will be asked to supply the name of the directory. You must supply a complete pathname if the directory is not a subdirectory of your working directory.

Example 1:

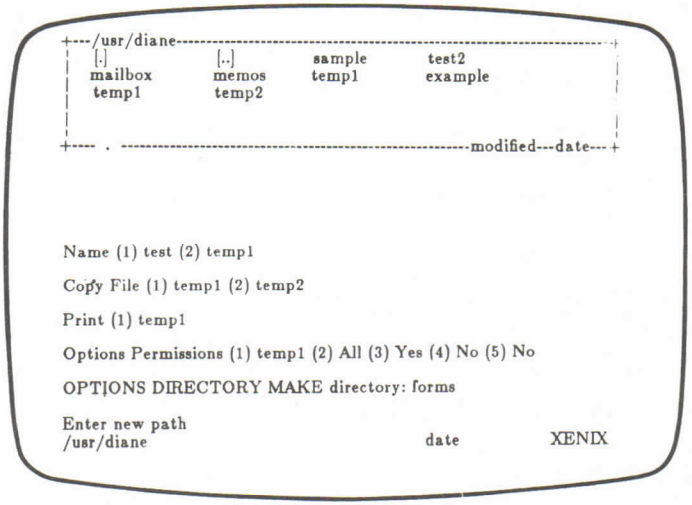
Assume that your working directory is named */usr/diane* . You want to create a "forms" directory under "diane". Follow these steps:

1. Choose the **Options** command from the main command menu.

2. Choose the **Directory** command.

3. Choose the **Make** command from the **Directory** menu.

4. Fill in the **Directory Make** menu as follows:



5. Press **CR** .

The message:

1 directory created

will be displayed in the command output area when XENIX has created the directory.

Example 2:

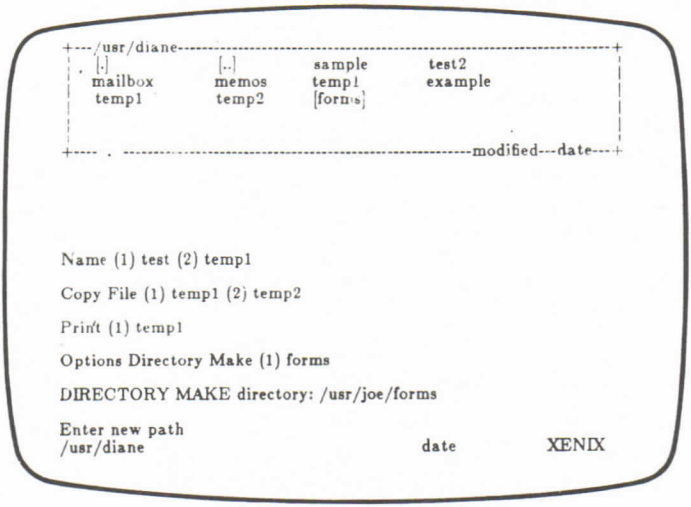
Assume that your working directory is "/usr/diane." To create a *forms* directory in "/usr/joe", follow these steps:

1. Choose the **Options** command from the main command menu.

2. Choose the **Directory** command.

3. Choose the **Make** command from the **Directory** menu.

4. Fill in the **Directory Make** menu with the complete *pathname* of the directory you want to make. Your screen should look like this:



5. Press **CR** .

Transferring to Other Directories: the View Command

If you want to change to another directory and look at the files in that directory, use the **View** command to transfer to the new directory. To use the **View** command, choose **View** from the main command menu. Type the name of the directory you want to transfer to in the "name:" command field. When you press **CR** , your working directory will be the directory you selected. The status line will change to reflect that you are now working in a different directory.

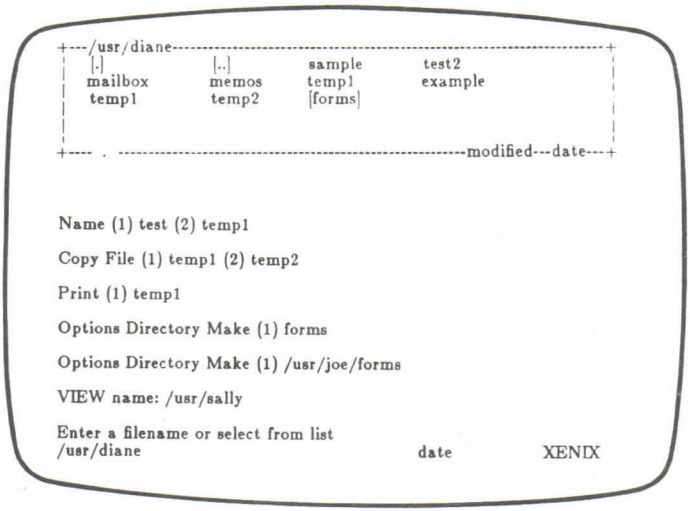
When you learn to use the Visual Shell window, you can use special keys to transfer to various directories. See the section entitled "Using Advanced Features" for more about changing directories.

Example:

Assume your working directory is "/usr/diane". You want to change to a directory named "/usr/sally". Follow these steps:

1. Choose the **View** command from the main command menu.

2. Type "/usr/sally" in the "name:" command field. Your screen should look like this:



3. Press the **CR** key.

Your working directory will now be "/usr/sally". To check this, look at the status line at the bottom of your screen. This line keeps track of your working directory.

Copying Directories: the Copy Directory Command

You can copy directories with the **Copy Directory** command. First, choose the **Copy** command from the main command menu. Then, choose **Directory** from the **Copy** menu. Type the *pathname* of the directory you want to copy in the "from:" command field. Type the new *pathname* of the directory in the "to:" command field. If you want all subdirectories of the directory copied also, choose "Yes" in the "recursive:" command field. The default is "No." When you press **CR** , your directory will be copied, and the message, **Copy Directory (1) filename (2) filename** will appear in the command output area of the screen.

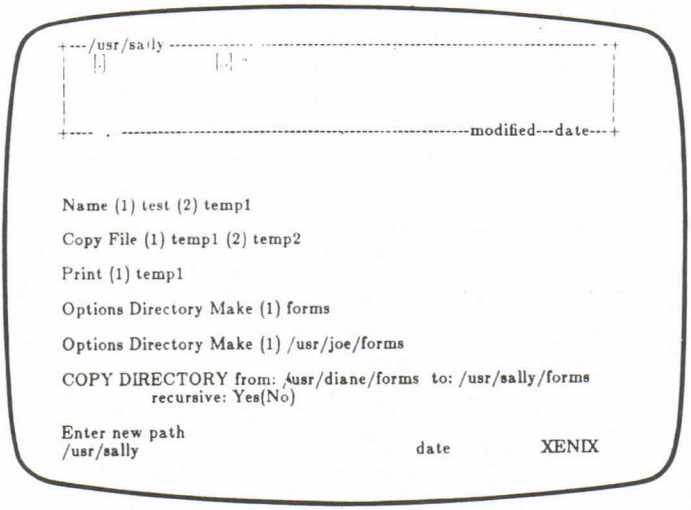
Example:

To make a copy of the "/usr/diane/forms" directory and name the directory "/usr/sally/forms", follow these steps:

1. Choose **Copy** from the main command menu.

2. Choose **Directory** from the **Copy** menu.

3. Fill in the command fields so that your screen looks like this:



4. Press **CR** .

You now have two identical directories:

- */usr/diane/forms* .
- */usr/sally/forms* .

Deleting Directories: the Delete Command

You can delete directories with the **Delete** command. First, choose the **Delete** command from the main command menu. Type the *pathname* of the directory you want to delete in the "name:" command field. When you press **CR** , the directory will be deleted. The message, **Delete (1) pathname** will be displayed in the command output area of the screen.

You cannot delete a directory unless it is empty (contains no files).

Example:

Assume that you have a */usr/tmp/diane* directory on your disk. To delete the */usr/tmp/diane* directory, follow these steps:

1. Delete each file from the directory using the **Delete** command on the main command menu.

2. When the directory is empty, choose **Delete** from the main command menu.

3. Type */usr/tmp/diane* in the "name:" command field.

4. Press **CR**

The */usr/tmp/diane* directory is now deleted.

Setting Directory Permissions: the Options Permissions Command

You can set different directory permissions with the command. To set a permission on a directory, follow the same steps for files, as described earlier in this section. Type the pathname of the directory you want to protect in the "name:" command field. When you press **CR** , the protection for your directory will have changed.

LISTING PERMISSIONS: THE RUN COMMAND

You can check the permissions that are set on both files and directories by using the Visual Shell Run command and "running" the XENIX **ls -l** (for list) command. Follow these steps:

1. Choose **Run** from the main menu.

2. When the **Run** menu appears, type:

   ls -l

   in the "file:" command field.

3. Press **CR** .

Your working directory and files will be displayed in the command output area of the screen. Protection values for each file and directory are also displayed.

MANAGING DISKETTES

The commands in this section will help you manage your diskettes. The commands discussed are on the Options FileSystem menu. They are: **Create**, **FilesCheck** , **SpaceFree** , **Mount** , and **Unmount**. The following table describes these commands:

COMMAND	TASK
Create	Creates a file system on a disk.
FilesCheck	Checks the files for bad sectors; fixes disks.
SpaceFree	Tells you how much space is left on the disk.
Mount	Allows you to "mount" (insert and use) a floppy disk.
Unmount	Allows you to "unmount" (take out) a floppy disk.

Creating a File System on a Diskette

You cannot mount a diskette unless it has a file system on it. If the diskette is new (right out of the box), it doesn't have a file system, so you must create one before mounting it. Use the **Options FileSystem Create** command to create a file system on a diskette.

To create a file system disk on a diskette, follow these steps:

1. Insert a formatted diskette into your system's diskette drive.
2. Choose the **Options FileSystem Create** command.
3. Fill in the appropriate fields.
4. Press **CR** .

Warning: If you create a file system on a diskette, then any data previously stored on that diskette is destroyed.

Example:

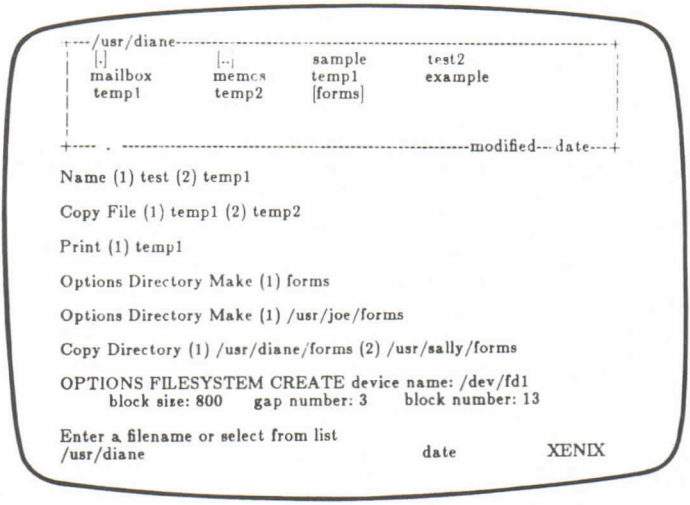
The following example creates a file system on a diskette. A file system is created on the diskette in the diskette drive named /dev/fd1. The diskette is then mounted.

1. Insert a formatted diskette into drive 1 *(/dev/fd1)* .
2. Choose **Options** from the main command menu.

3. Choose **FileSystem** from the **Options** menu.

4. Choose **Create** from the **FileSystem** menu.

5. Create a file system named "/dev/fd1" on the diskette by filling in the command fields as follows:



6. Press **CR** .

Mounting a Diskette

To mount a diskette on your computer, follow these steps:

1. Create an empty directory in your working directory. A common name for this directory is *mnt* . If you already have an empty directory, you may use that directory instead. (For information on making directories, see the subsection entitled "Creating a Directory: The Options Directory Make Command" earlier in this section.)

2. Insert a diskette containing a file system into your system's diskette drive.

3. Mount the diskette with the **Options FileSystem Mount** command.

You are now ready to use the diskette. If the mount is unsuccessful, use the **Options FileSystem FilesCheck** command to check the diskette.

Example:

You can mount the contents of the diskette's file system in an empty directory with the **Mount** command. Follow these steps:

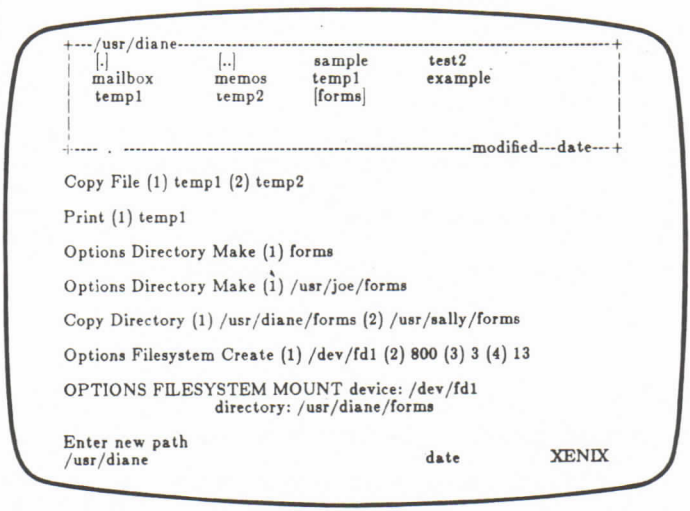
1. Insert a diskette with a file system on drive 1 (*/dev/fd1*).

2. Choose **Options** from the main command menu.

3. Choose **FileSystem** from the **Options** menu.

4. Choose **Mount** from the **FileSystem** menu.

5. Mount the diskette (*/dev/fd1*) in the empty directory you created (*/usr/diane/fd1*) by filling in the command fields so that your screen looks like this:



6. Press **CR** .

You can use the diskette simply by transferring to the */usr/diane/fd1* directory with the **View** command. If you do not know how to transfer to another directory, see the section entitled "Transferring to Other Directories: The View Command" earlier in this section.

Unmounting a Diskette: the Unmount Command

To remove a mounted diskette from a diskette drive, you must first "unmount" the diskette. That is, you must break the connection between the operating system and your diskette. You do this with the **Options FileSystem Unmount** command. Follow these steps carefully:

1. Transfer to a directory other than the "mounted" directory. For example, transfer to the */usr/diane* directory. Use the **View** command to transfer to a different directory.

2. Choose **Options** from the main command menu.

3. Choose **FileSystem** from the **Options** menu.

4. Choose **Unmount** from the **FileSystem** menu.

5. Type the diskette drive name in the "device name:" command field.

6. Press **CR** .

7. Take out the diskette.

Checking Diskettes: the Filescheck Command

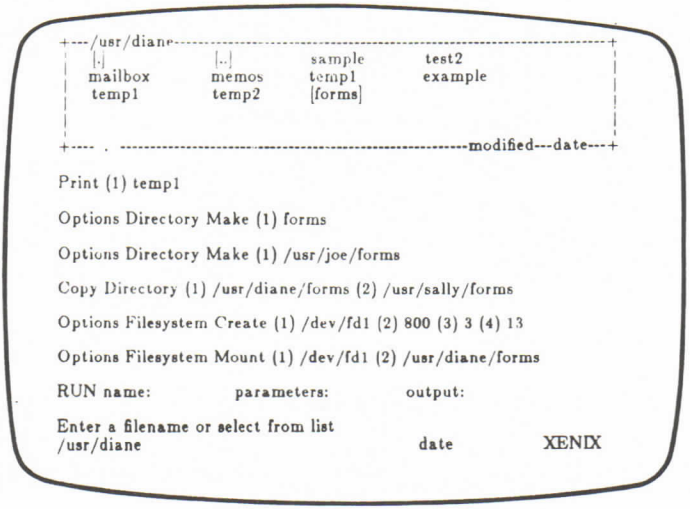
If you cannot mount a diskette, it is wise to run it through **FilesCheck.** Refer to the section entitled "Command Directory" for more information on the **Options FileSystem FilesCheck** command.

To check a diskette, choose **Options** from the main command menu. Next, choose the **FileSystem** command. A **FileSystem** menu will appear. Choose **FilesCheck.** Fill in the "device:" command field with the name of the diskette drive. When you press **CR** , the Visual Shell reports on the status of the diskette, and fixes any problems (such as bad sectors). Using **FilesCheck** may remove data from damaged portions of the diskette, making the data inaccessible.

RUNNING APPLICATIONS: THE RUN COMMAND

The **Run** command helps you run applications. These applications can be batch files that you create with an editor (using the **Edit** command), spreadsheet programs such as Microsoft Multiplan, and high-level language programs, such as C.

When you choose the **Run** command, the following **Run** menu appears:



Since the **Run** command can be used for many different files and applications, the **Run** menu provides space to tell the operating system what you want to run and how to run it. The "name:" command field is used to specify which program or command to run. The "parameters:" field is used to specify any arguments that the operating system may need to run the program or command. Any special switches for the command or program should be typed in the "parameters:" command field.

You can use the "output:" field to direct output from the application or command to another file or device; or you can access the **Filter** menu by placing a bar (|) symbol in this field. (See below for more information on the **Filter** menu.) When you press the **CR** key, your application will run. When you exit your application, the main command menu is redisplayed.

Using Filters

A filter is a command that reads your input, transforms it in some way, and then outputs it, usually to your terminal or to a file. In this way, the data is said to have been "filtered" by the program. Since filters can be put together in many different ways, a few filters can take the place of a large number of specific commands. Visual Shell filters include: **Count** , **Get** , **Head** , **More** , **Sort** , and **Tail.** The following list describes these filters. The corresponding XENIX command name is enclosed in parentheses. If you need more information on these commands, look in the XENIX User and System Administrator Reference Manual under the XENIX command name.

You can access the Visual Shell filters by placing a bar (|) symbol in the "output:" command field of certain command menus. The **Filter** menu will appear. When you choose an appropriate filter and press **CR** , your program or command will be piped through that filter. Example 2, below, illustrates how to access the **Sort** filter.

For more information on piping XENIX commands, refer to the first volume of this Guide. The **Filter** menu includes a **Run** command. This command is used to access filters you have written. To run a command through your own filter, choose the **Run** command from the main command menu. Type the name of the command in the "name:" command field, and type a bar symbol (|) in the "output:" field. Select the **Run** command on the **Filter** menu. Type the filename associated with your filter in the "name:" field. When you press **CR** , the command will be piped through your filter. Example 3 illustrates this process.

Three examples are provided below. The first example illustrates how to start a software application package, Microsoft Multiplan, with the **Run** command. The second example shows you how to access a XENIX command not on the command menu and how to use the **Filter** menu. Example 3 describes how to pipe a XENIX command through a filter that you have written.

Example 1: Running an Application

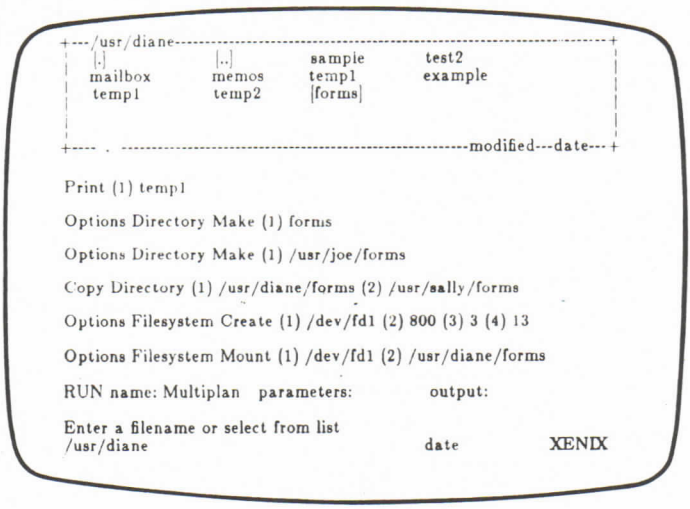
To run Microsoft Multiplan, follow these steps:

1. Make sure that Multiplan exists on your computer. If it is on a diskette, mount the diskette before proceeding to step 2.

2. Choose the **Run** command from the main command menu.

3. When the **Run** command menu appears, type "Multiplan" in the "name:" command field. Your screen should look like this:

FILTER	TASK
Count	Counts lines, words, and characters (wc).
Get	Searches files for a pattern (grep).
Head	Displays the first few lines of a file (head).
More	Displays files 23 lines at a time (more).
Sort	Sorts files (sort).
Tail	Displays the last part of a file (tail).



4. Press the **CR** key.

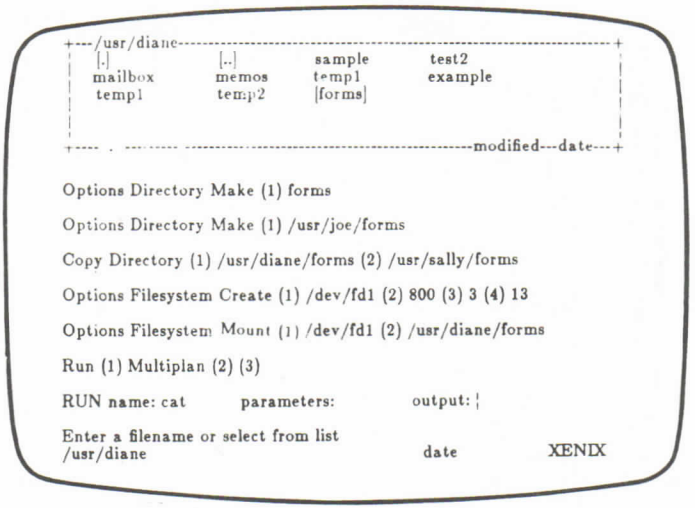
Multiplan will appear on your screen. When you exit Multiplan, the Visual Shell main command screen will be redisplayed.

Example 2: Running a Command With the Filter Menu

You can sort a list of names and send the sorted output to a file named *myfile* by using the **Run** command with the **Sort** filter. To create a list of names, you'll invoke the XENIX **cat** command using **Run.** The **cat** command copies keyboard input directly to a filter or file. Follow these steps:

1. Choose the **Run** command from the main command menu.

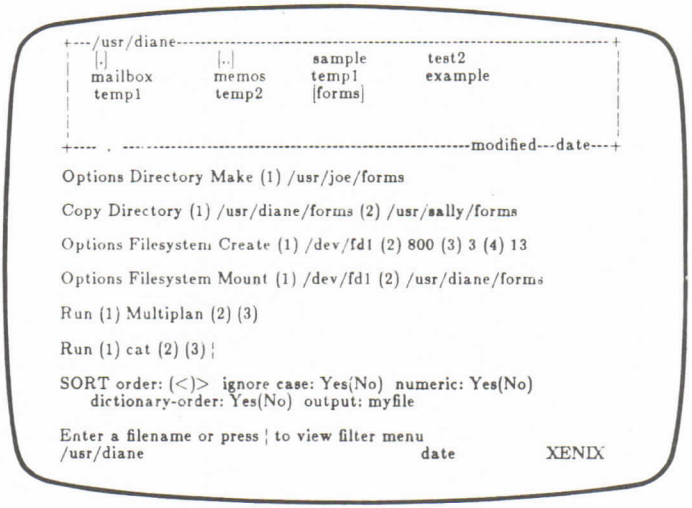
2. Fill in the **Run** command menu with the word "cat" in the "name:" command field. Go to the "output:" field and place a bar (|) symbol in it. Your screen should look like this:



3. Press the **CR** key. The **Filter** menu will appear.

4. Choose the **Sort** filter. The Visual Shell will display the **Filter Sort** menu. Since an A-Z sort is the default, you do not need to change this command field.

5. Go to the "output:" field. Type the filename *myfile* . Your screen should look like this:



6. Press the **CR** key. The main command menu will reappear, and a message similar to "Run (1) cat (2) (3) | (1) > (2) myfile" will appear in the command output area of the screen.

7. Now type the lines:

```
fred
larry
albert
snowden
george
```

8. To end the test, type **CTRL D** on a new line. Once you've ended the list, **cat** passes the names to the **sort** filter which sorts them alphabetically and places them in *myfile* . Use the **View** command to examine the *myfile* file.

Example 3: Using Your Own Filters

Assume that you have created a filter named unique that takes a sorted file, deletes duplicates, and displays the file without the duplicate entries. This filter is stored as a file called *unique.s* in your working directory. The file you want to view is named *beta* , and is not sorted. To access the filter, follow these steps:

1. Choose **Run** from the main command menu.

2. Type the XENIX **sort** command in the "name:" field.

3. Type "beta" in the "parameters:" command field.

4. Type a bar symbol (|) in the "output:" field to access the **Filter** menu.

5. Select **Run** from the **Filter** menu.

6. Type "unique" in the "name:" command field to access your filter.

7. Press **CR** .

The Visual Shell will display the beta file, sorted and without duplicate entries, on the screen.

SENDING AND RECEIVING MAIL

You send and receive mail with the **Mail** command on the main command menu. When you choose the **Mail** command, a **Mail** menu appears. There are two commands on the **Mail** menu: **Read** and **Send.** Use the **Read** command to read any electronic mail you receive; use the **Send** command to send messages to other users of the system.

As soon as you select Mail, you enter the mail system.

There are several commands you can perform while you are in **mail** ; deleting messages, saving messages in a file, editing a message you are composing, and canceling a message. These commands are discussed in the examples below. For more information about **mail** , see the appropriate chapter in this guide.

Example:

To read mail:

When you see the message **Mail** on the status line, choose the **Mail** command. A **Mail** menu will appear. Choose **Read.** A list of your mail messages will appear on the screen. To read message 1, type 1 and press **CR** . To read message 3, type 3 and press **CR** . When you have finished reading your mail, press **CTRL D** to return to the main command menu.

To delete a message:

To delete a message from your mailbox, type:

d *number*

where *number* is the number of the message you wish to delete. If you decide you want to save a deleted message, you can restore it before you quit **Mail** by typing:

u *number*

where *number* is the number of the message you want restored. You cannot

restore a message after you have exited **mail**.

To save a message in a file:

To save a message in a file, type:

s *number pathname*

where *number* is the number of the message being saved, and *pathname* is the pathname of the file in which you want to save the message.

To edit a message:

If you wish to edit a message you are composing, you can enter the **vi** editor from **mail** by typing:

~e

on a line by itself. See the first volume of this guide for information on how to use **vi**.

When you exit the editor you will be back in **mail**.

To cancel a message.

If, while composing a message, you decide not to send it, you can cancel the message by pressing the **INTERRUPT** key twice. After you have pressed **INTERRUPT** twice, you will return to the **Mail** menu.

To send mail:

1. Choose **Mail** from the main command menu.

2. Choose the **Send** command.

You are now in the XENIX mail system. To send mail, type the user's name in the "to" command field. Press **CR** . Next, type a message. When you press **CTRL D** , XENIX will send your message to the specified user or users.

To exit Mail:

If you are reading mail, press **CTRL D** to exit **mail** and return to the main command menu. If you are sending mail, you will return to the main command menu automatically; don't press **CTRL D** again or you will log yourself out.

Example:

This example sends the message "Hello there" to users Sue and Joe.

1. Choose **Mail** from the main command menu.

2. Choose **Send.**

3. Type "Sue" and "Joe" in the "to:" command field.

4. Fill in the "Subject:" and "Cc:" (Carbon copy) fields, pressing **CR** after each.

5. Type "Hello there" in the message area.

6. Press **CR** .

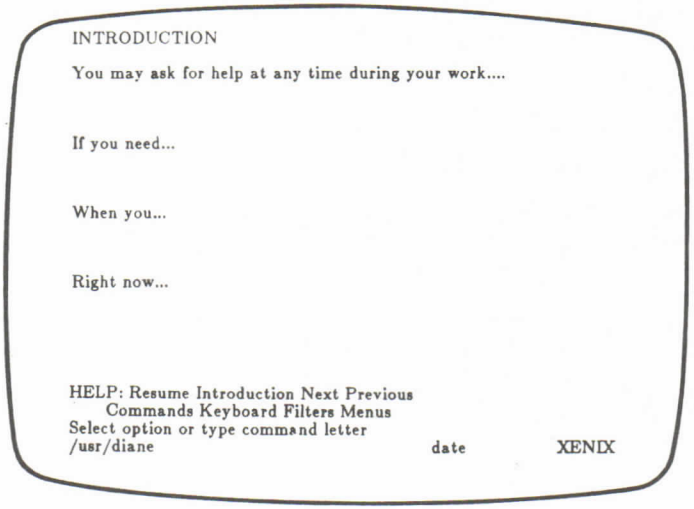
7. Press **CTRL D** .

When you exit the mail system, you will be returned to the XENIX Visual Shell.

GETTING HELP: THE HELP COMMAND

The Visual Shell includes a special **Help** command to assist you while using the XENIX operating system. This **Help** information is always available to you.

To access Help text, the **Help** command from the main command menu. You will see a **Help** menu that looks like this:



Use this menu to view various parts of the Help information. If you type "C" (for **Commands**), a Command Overview appears on the screen. This describes how to choose commands with the Visual Shell. When you type "N" (for **Next**), more Command Overview Help text is shown. You can use the **Next** command when the Help text is longer than one screen.

The following list describes the Help commands:

Resume	Resumes state of screen before you asked for help.
Next	Moves **Help** text forward by one screenful.
Previous	Moves **Help** text backward by one screenful.
Introduction	Moves you to the beginning of **Help** text.
Commands	Gives you Visual Shell command information.
Keyboard	Tells you how to use the keyboard.
Filters	Describes operating system filters and command piping, and how to use filter menus.
Menus	Illustrates how to change command menus.

You can access Visual Shell information with the **Help** command. For example, to find out which keys perform which functions, select the **Help** command, then type "K" (for **Keyboard**). The beginning of the list of keys appears. Type "N" (for **Next**) to view the rest of the list.

You can also access the **Help** command by pressing the question mark (?) key while you are selecting a Visual Shell command or filling in a command field. The example below illustrates this use of the **Help** command.

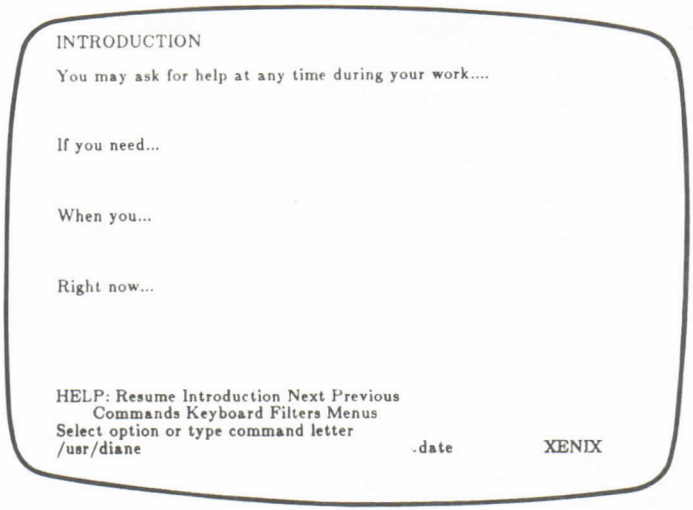
Example:

To access Help text on the **Copy** command, follow these steps:

1. Choose the **Copy** command by pressing **SPACE** ; do not type "C" and do not press **CR** .

2. When the **Copy** menu appears, press the **HELP** key.

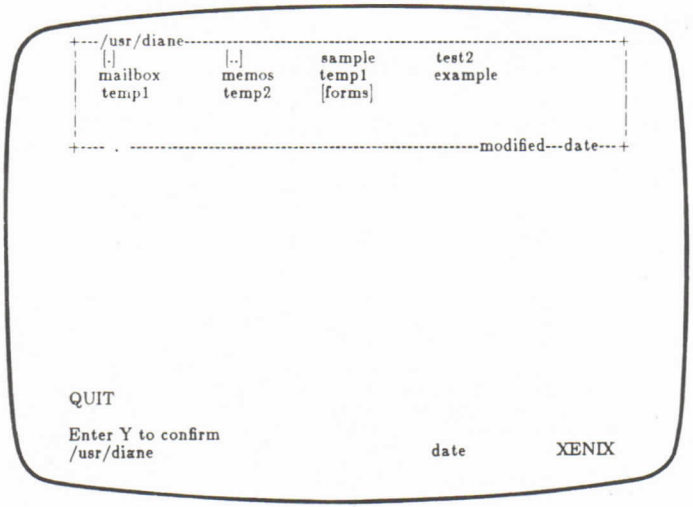
The screen display will be replaced by Help information on the **Copy** command. Your screen should look like this:



The information on the **Copy** command describes what happened when you copied the file *temp1* to *temp2* earlier in this manual. The **Help** menu appears at the bottom of the screen. You can use the **N** (for) command to view more of the Help text, or you can choose any of the subjects that you want to learn about. When you press **R** (for **Resume**), the main command menu is redisplayed.

QUITTING

The **Quit** command is used to quit the XENIX session. To quit, choose the **Quit** command on the main command menu. Your screen will look like this:



The Visual Shell asks you to confirm that you want to quit. To quit, type "Y" (for Yes). If you press any other character or number, the Visual Shell will return to the main command menu. When you type "Y", the screen will clear and you will have exited the Visual Shell. To restart the shell, press the **CR** key.

Example:

To quit the session, follow these steps:

1. Choose the **Quit** command from the main command menu.

2. The **Quit** screen will appear. It asks you to **Type Y to confirm:** Type "Y".

You have now exited from the Visual Shell.

USING ADVANCED FEATURES

INTRODUCTION

The Visual Shell is more than just an operating system interface; it is designed to help you enter commands and perform functions efficiently. Two Visual Shell features give you flexibility and power: the window and a modifiable menu structure. You will learn about the window and how to change Visual Shell menus in this section.

USING THE WINDOW

The Visual Shell can set aside a *window* at the top of your screen in which you can display and access information. You can manipulate directories and is window. The window can be permanent or temporary. A permanent window is one that has been explicitly opened and will remain open (with necessary redrawing) until explicitly closed. This is accomplished with the Window command. For information on opening permanent windows on the screen see the subsection entitled "Expanding the Window: The Window Command". A temporary window will disappear after the next command is processed.

Showing the Directory

Just as you can view text files in the window, you can use the direction keys on your keyboard to view directories in the window. (Refer to the section entitled "Key Directory" for information on the direction keys **UP** , **DOWN** , **LEFT** , and **RIGHT** .) When you use one of these keys, a temporary window appears on your screen and remains only until you execute a Visual Shell command.

To display a directory, press one of the direction keys when the main command menu appears on the screen. A temporary window will be drawn in the upper portion of the screen.

The window contains a directory listing for your working directory. Two kinds of information are shown: names of files and names of directories. Directories and subdirectories are shown in brackets ([ ]), and each executable file is preceded by an asterisk (*). Text files are not marked in any special way.

You can use the direction keys to select filenames and directory names in the window. The bottom border of a window always contains information about the selected directory or file. The border tells you the name of the directory (or file), its size in bytes, and the date it was last modified. For example:

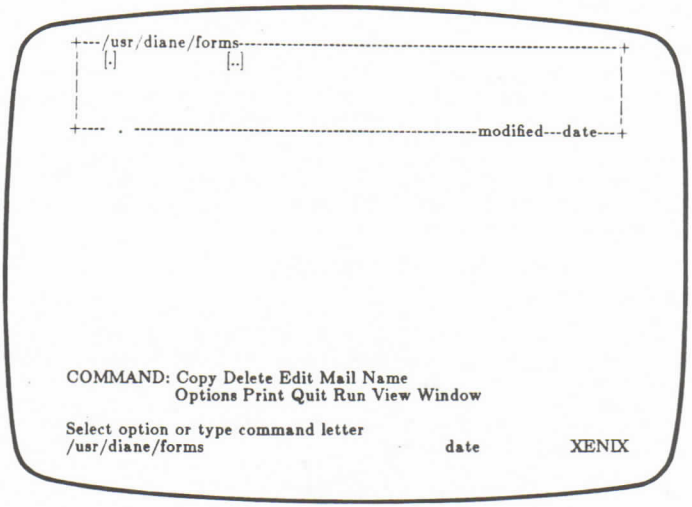
_____ myfile _____ 621 bytes _____ modified Aug. 2, 1983

Two special keys are used with the window. These keys are the equal sign = and the hyphen - . For simplicity, we will call these the **SHOW** and **GOAWAY** keys. If the cursor is highlighting a directory, you can press the **SHOW** (=) key to see the listing for that subdirectory displayed in the window. Pressing the **GOAWAY** (-) key returns the display to the parent directory. You can always access a

parent directory by pressing the **GOAWAY** key. Using these two keys, you can change your current working directory. This is another way to perform a **View** command on a directory. Notice that there are several subdirectories in your working directory. To view the directory named */usr/diane/forms* , follow these steps:

1. Use the direction keys to select the ["forms"] subdirectory name.

2. Press the **SHOW** (=) key to view the */usr/diane/forms* directory. Your screen will look like this:



3. Press the **GOAWAY** (-) key to redisplay the /usr/diane directory in the window.

Showing Text Files

You can also view text files in the window with the **SHOW** and **GOAWAY** keys. To view a text file, press one of the direction keys so that the working directory is displayed in the window. To view a text file, select the name of the file you want to view. Press the **SHOW** key. The directory will disappear and be replaced by the text file you selected. You can use the scrolling keys to view the file. (Refer to the section entitled "Key Directory" for information on keys that scroll through files.) When you press the **GOAWAY** key, the working directory is redisplayed in the window.

Note

You can only show" text files that reside in the directory that is currently displayed on the screen. To display other text files, either change directories or use the **View** command and specify a full pathname.

Example

To view the *templ* file in your working directory, follow these steps:

1. Press one of the direction keys when you see the main command menu on the screen. Your working directory will appear in the window.

2. Select the filename *templ* .

3. Press the **SHOW** key.

The *templ* file will be displayed in the window. To return to your working directory, press the **GOAWAY** key.

EXPANDING THE WINDOW: THE WINDOW COMMAND

Windows that appear in response to pressing a direction key are temporary windows that disappear after the next command is pressed. The **Window** command can be used to create a permanent window on the screen, and to increase the size of the window.

To create a permanent window that is larger than the default five-line window, choose the **Window** command from the main command menu. The **Window** menu will appear. This menu asks you whether you want to redraw the window after each command. The default is "No". To create a permanent window, go to the **redraw:** command field and choose "Yes". To establish a larger window, go to the height in lines: command field. Type a number from 6 through 16. (The window cannot be more than 16 lines long.) When you press the **CR** key, the main command menu will be redisplayed. A permanent window will appear on the screen until you close it with the **Window** command. sp To close the window, choose the **Window** command from the main command menu. Choose the "No" option in the **redraw:** command field. Press **CR** . The display will return to the main command menu, and the window will not appear until you press a direction key.

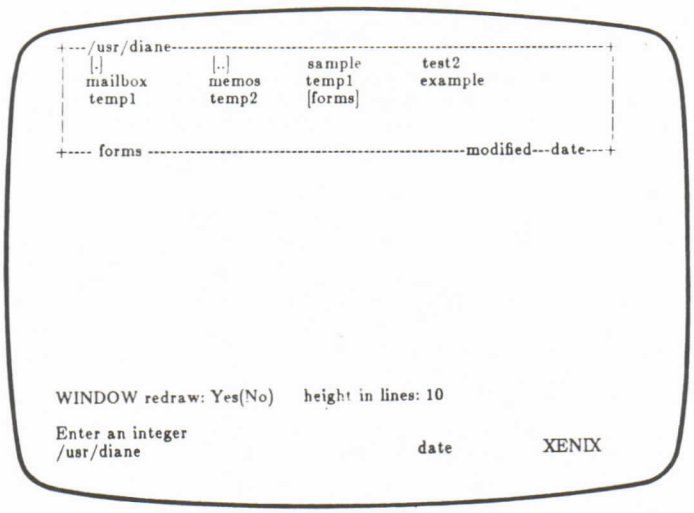
Example

To create a permanent 10-line window on the screen, follow these steps:

1. Choose the **Window** command from the main command menu.

2. Go to the redraw: command field and choose "Yes".

3. Go to the **height in lines:** command field. Type the value 10. Your screen should look like this:



4. Press the **CR** key.

The main command menu will be redisplayed. A window 10 lines long will appear in the upper part of the screen. To close this permanent window, choose the Window command and go to the redraw: command field. Choose "No". When you press **CR** , the main command menu will be redisplayed and the window will be closed.

USING THE WINDOW WITH SIMPLE COMMANDS

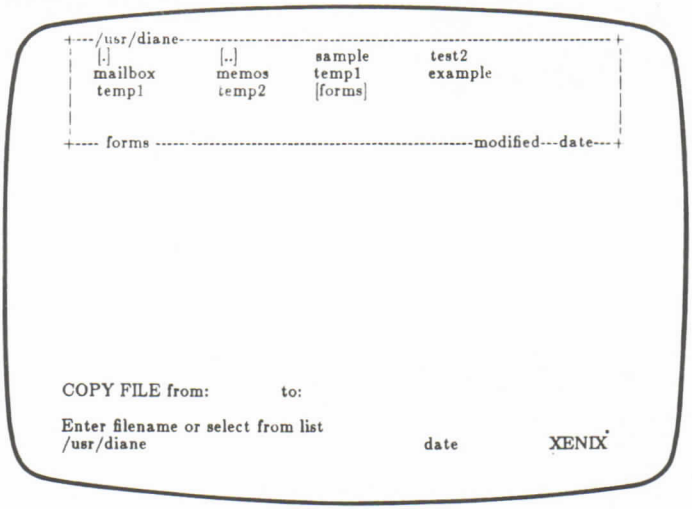
One of the advantages of using a window is that you can fill in command menus with filenames and directory names by selecting them in the window. The following example duplicates the **Copy File** command described in the the subsection entitled "Copying Files: The Copying File Command", when the *templ* file was copied to a file named *temp2* .

Example:

1. In the main command menu, type "C" (for Copy). The **Copy** menu will appear on the screen.

2. Select **File** from the **Copy** menu.

3. The cursor should be in the "from:" command field. Do not type the filename *templ* . Instead, press any direction key so that the working directory appears in the window. The following figure illustrates what your screen should look like:



4. Now, using the direction keys, select the filename *temp1* in the window. Notice that as you press the direction keys, the "from:" command field changes in the **Copy** menu. Each filename that is selected in the window appears in the "from:" field at the bottom of the screen.

5. When the name *temp1* appears in the "from:" field of the **Copy** menu, go to the "to:" field.

6. Type the filename *temp2* .

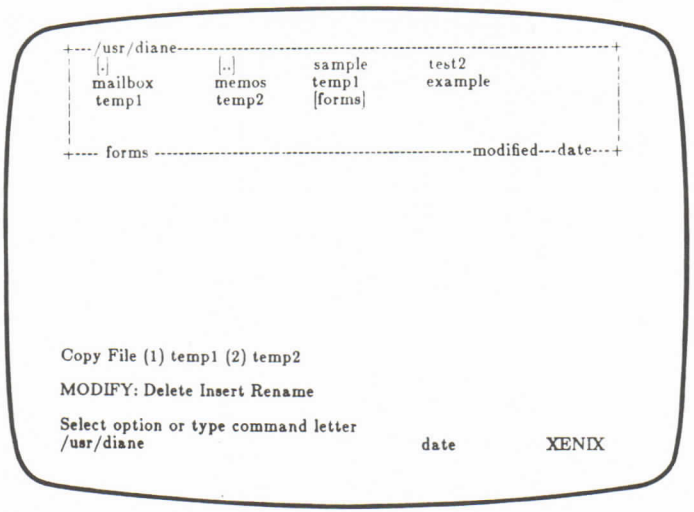
7. Press **CR** .

Output from the **Copy** command ("COPY (1) temp1 (2) temp2") will appear in the command output section of the command screen.

Both filenames and directory names can be selected to fill in command menus. Experiment with various commands by using the window to fill in the command menus.

CHANGING THE MENUS

You can modify menu commands (except those in **Help**) at any time. While in a menu (for example, the main command menu), you can change the commands on the screen by pressing the **MODIFY** (@) key. When you press @, the following **Modify** menu is displayed:



The choices on this menu are: **Delete** , **Insert** , and **Rename.** The following sections discuss what happens when you choose each of these commands.

Adding Commands to Menus

You can add commands by using the **Modify** menu. First, press the **MODIFY** key. Choose the **Insert** option. Fill in the command fields with the following:

1. The name of the command as you want it to appear on the menu.

2. The location of that command (you must give the command's full pathname). Refer to the section entitled "Command Mapping" for appropriate command names.

3. Where you want the command to appear on the menu.

   When you press the **CR** key, the command will be added to the main command menu.

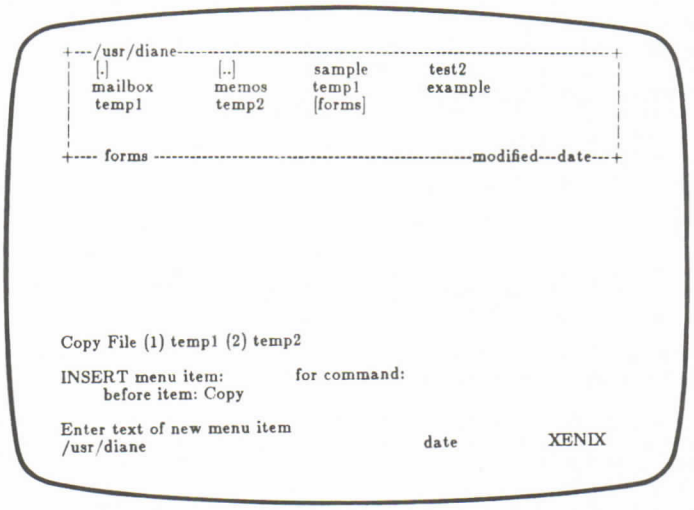
Example:

Assume that you want to add a **Look** command to the main command menu. This command will perform the same function as the XENIX **ls -l** (for "list") command. You want the command menu to contain 13 commands: **Copy, Delete** , **Text** , **Help** , **Look** , **Mail** , **Name** , **Options** , **Print** , **Quit** , **Run** , **View** and **Window.** To add the **Look** command, follow these steps:

1. When the main command menu is displayed, press the **MODIFY** (@) key. The **Modify** menu will appear.

2. Choose the **Insert** option.

3. The **Modify Insert** menu command will appear. This menu asks you what name you want to add to the main command menu and the command pathname that is associated with that name.



4. Type "Look" in the "menu item:" field.

5. Go to the "command:" field. Type the full pathname of the XENIX command for the value of the **Look** command.

6. Go to the "before item:" field. This asks you which command you want **Look** to come before on the menu. The Visual Shell automatically suggests **Mail** , the next item in the alphabetical list.

7. Press **CR** . The main command menu will include the new **Look** command between **Help** and **Mail** .

8. If you want to insert **Look** in a different place, press a direction key. The list of commands will appear at the top of the screen. Use direction keys to select the command you wish **Look** to precede.

Note

You must name your commands with unique first letters. Do not name your commands with letters that have already been used on the main command menu. If you choose a letter that is already taken, the Visual Shell will ask you to specify a different name for that command.

The Visual Shell recognizes all XENIX operating system commands and creates a submenu for each one you add to a command menu. To illustrate this, let's use the new **Look** command. In the command menu, go to the **Look** command and press **CR** . A special default command menu for your **Look** command will appear on the screen. This menu asks you to specify the parameters for the **Look** command. Type the name of the directory you want to view in the "parameters:" command field. When you press **CR** , you will be returned to the command screen and the directory you specified will appear in the window.

Note

If the directory you want to view is not a subdirectory of your working directory, you must type the *pathname* of the directory in the "parameters:" command field.

Deleting Commands From Menus

To delete a command from the main command menu, press the **MODIFY** key while in the command menu. (You do not have go to the command you want to delete.) The **Modify** menu will appear. Go to the **Delete** option and press **CR** . The **Modify Delete** menu simply asks you which command you want to delete. Type the name of the command you want to delete in the "menu item:" field. You can also use the direction keys to enumerate which command you want to delete. When you press **CR** , the main command menu will be displayed. It will no longer contain the command you just deleted.

Example:

To delete the **Look** command you just added to the main command menu, follow these steps:

1. Press the **MODIFY** key when the main command menu is displayed.
2. Choose the **Delete** option from the **Modify** menu.
3. Type the word "Look" in the "item to delete:" field.
4. Press the **CR** key.

The Visual Shell will redisplay the main command menu without the **Look** command.

Note

You cannot delete the **Help** command or any of the commands in the **Help** menu.

RENAMING COMMANDS

You can add and delete many commands with the Visual Shell. You can also change the name of any command on the menu. To do this, use the **Rename** command in the **Modify** menu.

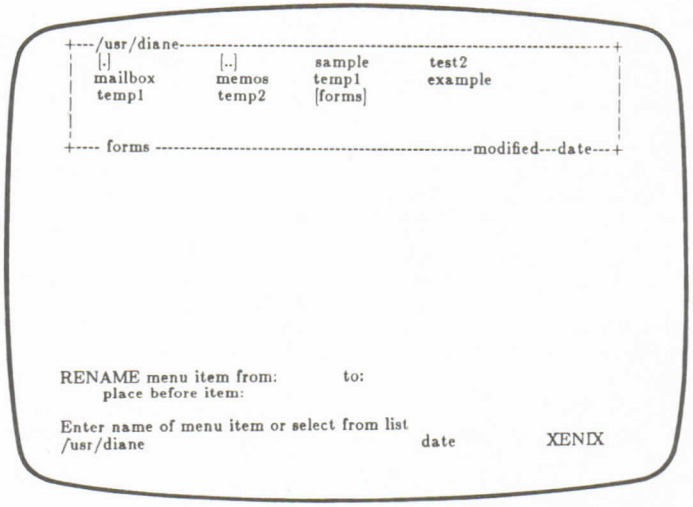
For example, you may want to rename the **Delete** command to "erase." (You might want to do this if you want to add a command that starts with the letter "D", which conflicts with the **Delete** command.) To rename a command, choose the **Rename** option in the **Modify** menu. The **Modify Rename** menu asks you which command to rename. Fill in the three command fields and press the **CR** key. The main command menu will be redisplayed and will include the new command name.

Example:

The following steps illustrate how to rename the **Delete** command to **Erase.**

1. While in the command menu, press the **MODIFY** key. The **Modify** menu will appear on the screen.

2. Go to the **Rename** command and press **CR** . Your screen should look like this:



3. There are three fields: "from:", "to:", and "before item:". Type "delete" in the "from:" field, and "erase" in the "to:" field. The Visual Shell will check to see if there are any commands in the main command menu that begin with the letter "E".

4. Go to the "before item:" field. The Visual Shell automatically suggests "Help" to place the renamed command in alphabetical order. Press **CR** to insert "Erase" before "Help".

5. The **Delete** command is now renamed **Erase**.

If you have chosen a letter that is already used on the menu, the Visual Shell prompts you to choose a different letter.

CREATING AND MODIFYING SUBMENUS

Creating and modifying submenus is easy with the Visual Shell. You must first decide what to name the main menu item. The command and its associated submenu must have the same name. Use the **MODIFY** key to display the **Modify** menu and choose the **Insert** menu.

There are three fields in the **Modify Insert** menu: "menu item:" , "for command:" and "before item:" . To add a submenu, type the name of the command in the "name:" field. Next, go to the "command:" field. Instead of typing a filename, type the word "Menu". This tells the Visual Shell that you are planning to add a submenu to the system. When you press the **CR** key, the Visual Shell will display the main command menu with the new top-level command.

So far, you have created a command with a submenu, but that has no commands for the submenu. To add commands to your new submenu, choose the top-level command and press **CR** . You should see the name of your submenu displayed on the screen. You can use the **MODIFY** key to call up the **Menu Modify** menu to add some commands to your submenu. Follow the steps described in the subsection entitled "Adding Commands to Menus" to add commands to the submenu.

Commands are deleted from a submenu the same way they are deleted from the main command menu. Refer to the subsection entitled "Deleting Commands from Menus" for more information. You cannot delete a submenu until each command has been deleted from it.

Example:

Assume that you want to be able to access several editors by using a submenu under a **Text** main command. These editors are named **Word** , **ed** and **vi** .

You must first add the main level command to the command screen using the **MODIFY** key and the Insert option. Follow these steps:

1. When the main command menu is displayed, press the **MODIFY** key. The **Modify** menu will appear.

2. Type "Text" in response to the "menu item:" field, and "Menu" in response to the "for command:" field.

3. Press **CR** . The main command menu is now displayed. The **Text** command has been added to this menu.

4. Choose the **Text** command and press **CR** . The screen will display "TEXT". Press the **MODIFY** key to display the **Modify** menu. Choose the **Insert** menu.

5. Add the first editor by typing the name of the editor (for example, Word) in the "menu item:" field and the location of the editor on disk (for example: */bin/word*) in the "for command:" field.

6. Press **CR** to display the main command screen.

7. Repeat steps 4, 5, and 6 to add **ed** and **vi** subcommands to the **Text** submenu.

To delete the submenu, first use the **Modify** menu to delete the commands **Word** , **ed** and **vi** . Next, return to the command screen and use the **MODIFY** key to display the **Modify** menu. Choose the **Delete** menu and type "text" in response to the "command:" field. When you press **CR** , both the submenu (named **Text**) and the top level **Text** command will be deleted.

CHANGING COMMAND MENUS

In the previous section, you learned how to add, delete, and rename commands. You also learned how to create and delete submenus. This section describes how to change and create command menus. Command menus are the menus that you fill in once you have selected a command.

.mnu FILES

All XENIX commands are actually executable files on disk. You can see the files when you do a directory listing for the directory that contains XENIX commands on your system (usually */bin*). Most XENIX command files have a companion called an *.mnu* file. All files have filenames which end with *.mnu* . An *.mnu* file contains the information necessary for the Visual Shell to create a command submenu. An *.mnu* file describes the following:

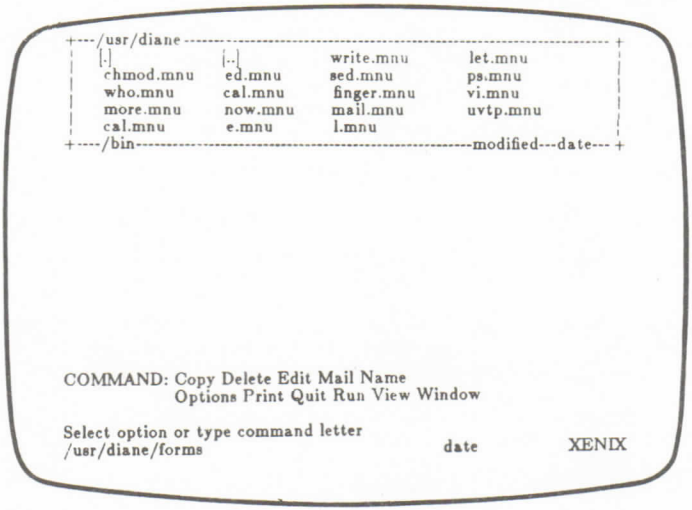
1. The submenu(s) for one operating system command.

2. The defaults and possible values of the fields within the submenu(s).

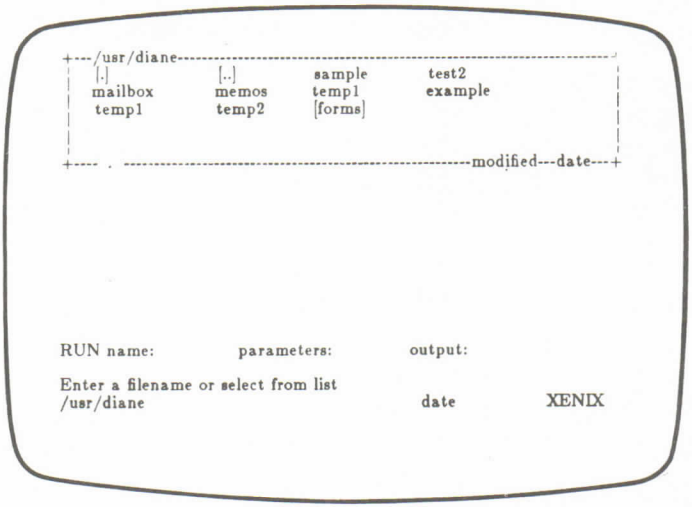
3. The actual command line arguments to be used once you press **CR** . These arguments act as a template into which values from the submenu are inserted.

4. Help text. This text appears on the screen if you press the **HELP** key (the ? key) while the command submenu is being displayed.

The *.mnu* files are stored in a separate directory called */bin* . Use the View command or the **SHOW** key to see which files have associated *.mnu* files. Notice that each command is matched with a file that ends with *.mnu* . Your screen should look similar to this:



If you add commands to the Visual Shell using the **MODIFY** key, you do not have to add a corresponding command menu. When you choose the new command, the Visual Shell first searches in the */bin* directory to find a *.mnu* file that corresponds to the command. (For example, a finger command might have an associated file.) If there is no associated *mnu* file for the command, the Visual Shell displays a default command menu. The default command menu looks like this:



You can create your own command menus by writing a *mnu* file that corresponds to a command that you have added to the Visual Shell. Refer to the section entitled "Making Your Own Menu Files" for information on how to create a *mnu* file.

KEY DIRECTORY

INTRODUCTION

In this chapter, the keys are referred to by their functional names (by what they do), rather than by what may be written on the key.

The following are the basic key assignments provided with your operating system. Depending on the keyboard you use, you may have additional function keys or different key assignments. An extended chart, which lists terminal-specific keys, is available in the Visual Shell by pressing the letter **H** (for Help) and then **K** (for Keyboard). (Press **N** , for Next, to view more of the chart.)

Note

These keys perform the following functions only in conjunction with Visual Shell menus. They may not work in the same manner if used during application programs or during operating system commands.

KEY CHART

KEY SEQUENCES	FUNCTION
CTRL E	UP DIRECTION. Moves the cursor up one line.
CTRL X	DOWN DIRECTION. Moves the cursor down one line.
CTRL S	LEFT DIRECTION. Moves the cursor left one character.
CTRL D	RIGHT DIRECTION. Moves the cursor right one character.
CTRL R E	PAGE UP. Moves the cursor up one page.
CTRL R X	PAGE DOWN. Moves the cursor down one page.
CTRL R S	PAGE LEFT. Moves the cursor left one page.
CTRL R D	PAGE RIGHT. Moves the cursor right one page.
CTRL Q	HOME. Moves the cursor to the beginning of the file or directory listing.
CTRL Z	END. Moves the cursor to the end of the file or directory listing.
CTRL C	CANCEL. Cancels present operation and returns to main command menu.
CR	CARRIAGE RETURN. Starts a command selected from a menu or carries out a completed command.
SPACEBAR	Selects the next item on a menu.
BACKSPACE , CTRL H	Selects the previous item on a menu. When editing responses in command fields, deletes selected characters.
TAB , CTRL I , CTRL A	TAB. Moves to and selects the entire contents of the next field in the command line.

KEY SEQUENCES	FUNCTION
CTRL Y , DELETE	DELETE. Deletes selected characters.
CTRL L	CHARACTER RIGHT. Selects the character to the right of the current character.
CTRL K	CHARACTER LEFT. Selects the character to the left of the current character.
CTRL P	WORD RIGHT. Selects the word to the right of the current word.
CTRL O	WORD LEFT. Selects the word to the left of the current word.
?	HELP. Requests information about the selected command or the command in progress at the time of the request.
=	SHOW. Displays directories and text files; displays submenus for commands in window.
-	GOAWAY. Returns window display to parent or current directory.
@	MODIFY. Displays the **Modify** menu.
!	REDRAW. Redraws the screen.
/	BAR. Displays the **Filter** menu.

COMMAND DIRECTORY

INTRODUCTION

This section describes the Microsoft Visual Shell commands. They are:

Copy Directory
Copy File
Delete
Edit
Help
Mail Read
Mail Send
Name
Options Directory Make
Options Directory Usage
Options FileSystem Create
Options FileSystem FilesCheck
Options FileSystem Mount

```
Options FileSystem SpaceFree
Options FileSystem Unmount
Options Output
Options Permissions
Print
Quit
Run
View
Window
```

The command descriptions explain the action of each command, the purpose of the command fields, and the meaning of the messages displayed by a command.

The two following definitions, which you will be using later on in this chapter, describe the required syntax for filenames and pathnames.

filename A *filename* is a string of up to 14 characters. Characters such as period (.) and hyphen (-) are allowed so you may create filename extensions that are used to identify types of files. An example of a *filename* with an extension is newfile.mnu. Refer to the first volume of this guide for more information on naming files.

pathname A *pathname* is a sequence of directory names followed by a simple *filename* , each separated from the previous one by a forward slash (/). *Pathnames* are used to uniquely identify files that may have the same name or that are not in your current directory. An example of a *pathname* is /usr/joe/testfile.

Copy

copy : File | Directory

The **Copy** command offers a choice of subcommands that perform the following tasks:

- Copy the contents of a file to a new file.
- Copy the contents of a directory to a new directory.

COPY DIRECTORY from: to: recursive: Yes (No)

Purpose

To copy the contents of the source directory to the destination directory.

Command Fields

from: Enter a directory name. The directory name must have the syntax described at the beginning of this chapter. If the directory specified in this field is not the working directory, make sure you supply an appropriate pathname.

to: Enter a directory name. The directory name must have the syntax described at the beginning of this chapter.

recursive: If you select "Yes" the Visual Shell will copy all subdirectories under the directory specified.

COPY FILE from: to:

Purpose

To copy the contents of the specified source file to the specified destination file.

Remarks

You may copy the contents of any existing file to the destination file. If the destination file already exists, the old contents are deleted before the new contents are copied to it.

If you copy the source file to another directory, you may give the destination file the same name as the source. Otherwise, you must use a different name.

Command Fields

from: Enter a filename. The filename must have the syntax described at the beginning of this chapter. If the file is

not in the working directory, make sure you supply an appropriate pathname.

to: Enter a filename. The filename must have the syntax described at the beginning of this chapter. If the file is not in the working directory, make sure you supply an appropriate pathname.

Delete

DELETE name:

Purpose

To perform the following delete operations:

- Delete a file from a directory
- Delete a subdirectory from a directory

Remarks

The command deletes a file from the current or specified directory, or deletes the directory specified by the pathname. If the file or directory does not exist, the command displays an error message.

Command Fields

name: Choose one of the following:

1. Enter a filename. The filename must have the syntax described at the beginning of this chapter. If the file is not in the working directory, make sure you supply an appropriate pathname.

2. Enter a directory name. It must have the syntax described at the beginning of this chapter. If the subdirectory is not in the working directory, make sure you supply an appropriate pathname.

Edit

EDIT filename:

Purpose

To load the system's interactive editor and edit the text file named after filename:

Remarks

The command relinquishes control to the system's interactive editor, so the command menu is erased from the screen and is replaced by the editor's own screen. The command menu is restored when you exit the editor.

Command Fields

filename: Enter a *filename* . The *filename* must have the syntax described at the beginning of this chapter. If the file is not in the working directory, make sure you supply an appropriate pathname.

Help

HELP: **Resume** | **Next** | **Previous** | **Introduction** | **Commands** | **Filters** | **Keyboard** | **Menus**

Purpose

To display information about the Visual Shell and its commands.

Remarks

The commands in the **Help** menu provide the following **Help** information:

Resume Exits **Help** and returns you to your previous screen.

Next Scrolls **Help** text one screenful ahead.

Previous Scrolls **Help** text one screenful backward.

Introduction Displays the beginning of Visual Shell **Help** text.

Commands Displays Visual Shell command information.

Filters Displays information on the Visual Shell filters **More** , **Sort** , **Head** , **Tail** , **Get** and **Count** .

Keyboard Displays information about the keyboard.

Menus Displays information on how to change the Visual Shell menus.

Mail

MAIL: Read | Send

Mail has two commands: **Read** and **Send**. These two commands are described in the following text.

MAIL Read

Purpose

To read mail sent by other users of the system.

Remarks

Once you have selected **MAIL Read** , you begin using the XENIX mail system to read your mail. Refer to the appropriate chapter of this guide for more information on the XENIX **mail** command.

Command Fields

There are no command fields.

MAIL SEND to:

Purpose

To send mail to other users on the system.

Remarks

After you have filled in the "to:" command field, you begin using the XENIX **mail** command. Refer to the appropriate chapter of this guide for more information on the **mail** command.

Command Fields

to: Type one or more user names in this command field and press **CR** . If more than one user name is specified, separate the names with a space.

Name

NAME from: to:

Purpose

To rename a directory or file.

Remarks

The new name must not be the same name of a file or directory already in the same directory.

Command Fields

from: Enter a file or directory name. The file or directory name must have the syntax described at the beginning of this chapter. If the file or directory is not in the working directory, make sure you supply an appropriate pathname.

to: Enter a file or directory name. The file or directory name must have the syntax described at the beginning of this chapter. If the file or directory is not in the working directory, make sure you supply an appropriate pathname.

Options

OPTIONS: Directory | FileSystem | Output | Permissions

Purpose

To offer a choice of subcommands to perform the following operations:

- Make a directory (under **DIRECTORY**).
- Determine disk usage (under **DIRECTORY**).
- Create a file system (under **FILESYSTEM**).
- Check a file system (under **FILESYSTEM**).
- Determine free space (under **FILESYSTEM**).
- Mount a floppy disk (under **FILESYSTEM**).
- Unmount a floppy disk (under **FILESYSTEM**).

- Set permissions for directories and files (under **PERMISSIONS**).
- Change Visual Shell command output on the screen (under **OUTPUT**).

The commands are described individually on the following pages.

Options Directory

The **Options Directory** menu has two commands: **Make** and **Usage** .

They are described below.

OPTIONS DIRECTORY MAKE directory:

Purpose

To make a new directory.

Command Fields

directory: Enter a directory name. The directory name must have the syntax described at the beginning of this chapter. The directory that you create will become a subdirectory of your working directory.

OPTIONS DIRECTORY USAGE name:

Purpose

To determine the number of blocks contained in all files and directories within each directory and file specified by "name:".

Remarks

This command does not count directories that cannot be read or files that cannot be opened. Files with holes in them will result in an incorrect block count.

Command Fields

pathname: Specify the pathname of a directory. The directory name must have the syntax described at the beginning of this chapter.

Options Filesystem

The **Options FileSystem** menu contains 5 commands:

COMMAND	PURPOSE
Create	Creates a file system.
FilesCheck	Checks and repairs files systems.
SpaceFree	Reports number of free disk blocks.
Mount	Mounts a file structure.
Unmount	Dismounts a file structure.

These commands are described below.

OPTIONS FILESYSTEM CREATE device: block size: gap no: block no:

Purpose

To create a file system.

Remarks

The block size, gap number, and block number command fields require numbers that are given in the "XENIX Installation and System Administration Guide".

Command Fields

device: Specify a device name, such as /dev/fd1. The device name must have the syntax described at the beginning of this chapter.

block size: Specify a block size, or the size of the created system.

gap no: Specify a gap number.

block no: Specify a block number.

OPTIONS FILESYSTEM FILESYSTEM device:

Purpose

This command is usually run when you have problems mounting a diskette on a machine. **Options FileSystem FilesCheck** audits and repairs inconsistent

conditions for file systems. If the file system is consistent, the number of files, number of blocks used, and number of blocks free are reported.

Remarks

Most corrective actions of this command result in some loss of data.

Command Fields

device: Specify a device name. The device name must have the syntax described at the beginning of this chapter.

OPTIONS FILESYSTEM MOUNT device: directory:

Purpose

To mount a diskette on the machine.

Remarks

This command tells the system that a removable file structure (i.e., a diskette) is present. The file structure is mounted on a directory. The directory must already exist; it becomes the name of the root of the newly mounted file structure. Refer to the **Options Directory Make** command for information on how to create a directory.

If you have problems mounting a diskette, refer to the **Options FilesSystem FilesCheck** command.

Command Fields

device: Specify the name of the device you are mounting, such as /dev/fd1. The device name must have the syntax described described at the beginning of this chapter.

directory: Specify the name of the directory in which the mounted file system will reside. The directory name must have the syntax described at the beginning of this chapter.

OPTIONS FILESYSTEM SPACEFREE device:

Purpose

Reports the number of free disk blocks.

Remarks

If the "device:" command field is not specified, **SpaceFree** reports free space on all mounted file systems.

Command Fields

device: Specify a device name. The device name must have the syntax described at the beginning of this chapter.

OPTIONS FILESYSTEM UNMOUNT device:

Purpose

Dismounts a mounted file system (diskette).

Remarks

Busy file structures cannot be dismounted with **Unmount.**

Command Fields

device: Specify the mounted device name. The device name must have the syntax described at the beginning of this chapter.

Options Output

OPTIONS OUTPUT commands like: VShell XENIX

Purpose

To set how the Visual Shell will display command syntax when a command has run.

Remarks

Normally, the Visual Shell will display its commands in the command output area of the screen. For example, when you have run **Options Directory Make** to create a directory, the Visual Shell will display, **Options Directory Make...** when the command has processed. If you want to see which XENIX command is running, select the "XENIX" field. The Visual Shell will display **mkdir ...** instead of **Options Directory Make .**

Command Fields

commands like: VShell XENIX Choose "XENIX" to see actual XENIX commands displayed. The default is Visual Shell commands ("VShell").

Options Permissions

OPTIONS PERMISSIONS name: who: (All) Me Group Others
read: (Yes) No write: (Yes) No execute: (Yes) No

Purpose

To change permissions on a file or directory.

Remarks

The following permissions groups are established:

All	All users
Me	One user
Group	All users in a specified group
Others	All users other than group and me

Command Fields

name: Type the name of a file or directory. The filename or directory name must have the syntax described at the beginning of this chapter.

who: Enable the permissions group by selecting All, Me, Group, or Others.

read: Select "Yes" or "No". "Yes" is the default.

write: Select "Yes" or "No". "Yes" is the default.

execute: Select "Yes" or "No". "Yes" is the default.

Print

PRINT filename:

Purpose

To print a file or files on the system's lineprinter.

Remarks

The command adds the specified file or files to the end of the printer queue. The file is printed when it reaches the top of the queue.

The specified file(s) must be text files.

Command Fields

filename: Enter a filename. The filename must have the syntax described at the beginning of this chapter. If the file is not in the working directory, make sure you supply an appropriate pathname.

Quit

QUIT:

Enter Y to confirm:

Purpose

To leave the Visual Shell and return to the XENIX command shell.

Remarks

You must type the letter "Y" to confirm that you want to leave the Visual Shell. On exit from the shell, the system saves the current environment so you may return to the same point when you start the shell again.

Run

RUN name: parameters: output:

Purpose

To run the program or command specified by "name:".

Remarks

The specified file must be one of the following:

- An application program
- A XENIX file that is executable
- The name of a XENIX command

If you specify a filename, the filename must have the syntax described at the beginning of this chapter. If the file is not in the working directory, make sure you supply an appropriate pathname.

Command Fields

name: Enter a filename, program name, or command name.

parameters: Enter any arguments to the program or command, including switches.

output: Specify a filename or device name to redirect output from the program or command. Specify a bar (|) symbol to access the Filter menu.

View

VIEW name:

Purpose

This command loads a file or directory into the window.

Remarks

If a file is loaded in the window, you can view the file by using the scrolling keys. If a directory is loaded in the window, you can travel around the directory by using the **SHOW** and **GOAWAY** keys. Refer to the section entitled "Key Directory" for more information.

Command Fields

name: Enter a file or directory name. The file or directory name must have the syntax described at the beginning of this chapter. If the file or directory is not in the current directory, make sure you supply an appropriate pathname.

Window

WINDOW redraw: Yes (No) height in lines:

Purpose

To turn the window display on or off and set the window height.

Remarks

When the window redraw is "Yes", a permanent window appears at the top of your screen. The window contains the output from the **View** command and the **SHOW** and **GOAWAY** keys. When redraw is "No", a window only appears when specifically requested with a **View** command or by pressing a direction key. The window command field is initially "No".

You may use the "height in lines:" field to set the window height in lines. Initially, the height is 5 lines. Maximum height is 16 lines.

Command Fields

redraw: Choose an option. "Yes" turns the window display on. "No" turns it off.

height in lines: Enter a number. It may be any number from 1 to 16.

COMMAND MAPPING

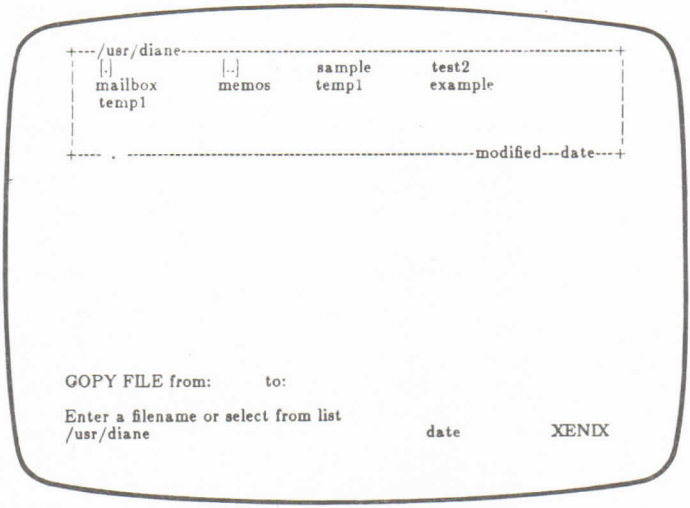
The following table maps all Visual Shell commands to XENIX commands.

XENIX COMMAND	VISUAL SHELL COMMAND
cp	Copy File
copy	Copy Directory
rm,rmdir	Delete
vi	Edit
--	Help
mail	Mail Read
mail	Mail Send
mv	Name
mkdir	Options Directory Make
du	Options Directory Usage
mkfs	Options FileSystem Create
fsck	Options FileSystem FilesCheck
df	Options FileSystem SpaceFree
mount	Options FileSystem Mount
umount	Options FileSystem Unmount
chmod	Options Permissions
--	Options Output
pr|lp	Print
CTRL D	Quit
--	Run
cat,cd	View
--	Window

MAKING YOUR OWN MENU FILES

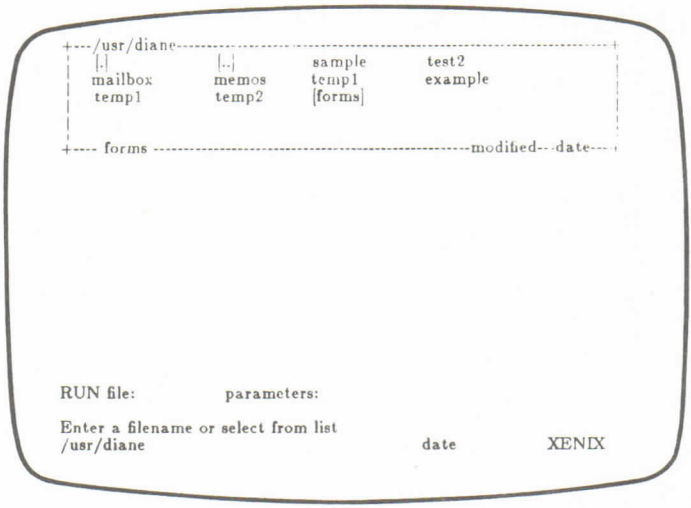
INTRODUCTION

When you use the Visual Shell, you select commands at the bottom of the screen. Each time you select a command, another screen appears. This screen is called a "command menu". For example, when you select the **Copy** command, a screen similar to the following appears:



A command menu contains fields that give information, such as a filename, to the command. To run the **Copy** command, you fill in the "from:" and "to:" fields and then press the **CR** key.

If the Visual Shell does not recognize the command you run (for example, if you have added a command to the menu), a default command menu appears for you to fill in. The default command menu looks like this:



COMMAND MENUS

Each Visual Shell command calls up a different command menu. Some command menus are defined in a special file named */bin/menu.def*. The command menus for user-added commands are stored in individual files with *.mnu* extensions, one for each command. For example, if you add a command named **Spread** (for Spreadsheet), you can define your own command menu with a file named *spread.mnu* . If the Visual Shell does not find a corresponding *.mnu* file for a command, it displays the default command menu.

.mnu FILES

A *.mnu* file describes:

1. Prompts that will be displayed on command menus.
2. Fields within command menus.
3. Operating system command-line arguments.
4. Help text for a command (optional).

You create a *.mnu* file by using an editor in XENIX such as **vi** , or by using the XENIX **cat** command. (See the "XENIX User and System Administrator Reference Manual" for more information on the **cat** command.) All *.mnu* files should be stored in the */bin* directory. The bin directory must be a defined search path.

COMMAND MENU PROMPTS

If you want the command menu to contain prompts, they should be inserted in the *.mnu* file just as they will appear on the screen. You may need to insert special characters to delimit fields. See the example *.mnu* files at the end of this appendix for more information.

COMMAND MENU FIELDS

There are two types of fields in a command menu:

1. Menu fields: Field selection is with a menu and you choose one of the options. Example:

   who: Me All Group Others

2. Fill-in fields: The response is text and you type a response, such as a filename. Example:

   filename:

A *.mnu* file contains a description for each field, whether it is a menu field or a fill-in field. Each field description describes the field's type, default value, and any other type-specific information. The following is a list of field types:

FIELD TYPE	DESCRIPTION
SELECT	Selection is from a menu.
FILENAME	Allows directory window selection and text.
FILENAMELIST	Allows directory window selection and text.
PARAMETERS	This is a text field.
OUTPUT*	This field is for output information.
NUMBER	This is an integer field (number).
DIRECTORY	Allows directory window selection and text.

OUTPUT* fields determine redirection of I/O. If used, the Visual Shell adds the necessary symbols, commands, and file names to the command line that is processed by XENIX. SELECT field types must always be followed by a series of text strings, one for each option in the field. These are generally used to specify switches.

Each field type has a fixed prompt that will appear at the bottom of the menu when the field is selected; for example, **Enter a filename** .

You can include a default response for each field type. That is, the Visual Shell can automatically display a response to a field when the command menu first appears. There are several different kinds of defaults for fields:

COMMAND LINES

After you have described the command menu prompts and fields, the *.mnu* file must contain a description of the command line. The Visual Shell uses this information to build the line that will be executed by XENIX. The command line description is a line with placeholders for variables that the Visual Shell will provide. The placeholders take the form ^x, where "x" is an integer that indicates the command menu field that supplies the value. The line appears as "exec = template" in the *.mnu* file, where "template" stands for the executable command line. See the *.mnu* files at the end of this appendix for examples of the "exec=" line.

HELP TEXT

Help text appears whenever you press the **HELP** key or use the **Help** command. This text will only be displayed if you ask the Visual Shell for help when choosing the top-level command or while in the command menu. You can type any text after the command argument line (exec=) in the *.mnu* file. Your text should begin after this line.

Help text must be separated from the *.mnu* file description by a line consisting of five hyphens.

EXAMPLES

The following examples show two typical *.mnu* files. The first *.mnu* file is named */bin/get.mnu* , and provides a unique command menu for the XENIX command **get** that a user has added to the top-level Visual Shell menu. The second *.mnu* file is named /bin/spread.mnu, and provides a unique command menu for a spreadsheet application that a user wishes to select from the top-level menu. Each *.mnu* file contains Help text.

Note

Both of these examples assume that the user has already added the commands **Get** and **Spread** to the top-level menu. See the section entitled "Using Advanced Features" for details on adding user commands to the Visual Shell.

Example 1: get.mnu

```
GET string:^^ filename:^^ linenumbers:Yes No
     casesensitive:^Yes No^ output:^^
fields
1 parameters
2 filename
3 select=2
"-n"
""
4 select=1
""
"-y"
5 output OPT
exec="fgrep ^3 ^4 ^1 ^2"
-----
```

GET finds a string in a file. If the "linenumbers:" command field is set to "Yes", GET reports the line numbers that contain the string. If the "casesensitive:" command field is set to "Yes", GET reports strings that exactly match the string and does not ignore case. You can direct output to a file or printer by specifying a filename or device name in the "output:" command field.

Typing a bar symbol (|) in the "output:" command field brings up the **Filter** menu.

Example 2: spread.mnu

```
SPREAD filename:^^
fields
  1 filename OPT
  exec="/bin/mp /bin/mp^1"
-----
```

SPREAD runs the electronic spreadsheet named Multiplan. Specify the filename of the worksheet in the field. If no worksheet is specified, Multiplan starts with a blank spreadsheet.

MESSAGE DIRECTORY

The Visual Shell displays the following messages on the message line:

Already at bottom of directory
Already at top of directory
At first character in field
At last character in field
Bad Help file
Cannot write to Help pointer file
Command is too long
Command not found
Default specifier not recognized
Enter Y to confirm
Enter Y to retry access to
Enter a filename or press | to view filter menu
Enter a filename or select from list
Enter an integer
Enter name of command
Enter name of menu item or select from list
Enter new path
Enter one or more filenames or select from list
Enter options
Enter text of new menu item
Error changing directories
Field has too many words
Field not optional
Field number greater than limit
Field number less than one
Field type not defined
File is empty
File not found
First letter conflict with existing menu item
Help file not available
Inappropriate field type for fill-in field
Inappropriate field type for menu field
Insufficient memory
Menu item not on current menu
Menu must be empty to be deleted
Missing "="
Missing command field type
Missing default specifier
Missing end quote
Missing field number
Missing quoted menu value descriptions
Missing sheet name in property sheet
Multiple responses are not allowed
Name list is empty
No character to delete
No character to delete
Not a valid character
Not a valid default for a menu
Not a valid default function
Not a valid integer
Not a valid key

```
Not a valid option
Not a valid path name
Path is too long
Permission denied
Select option
Select option or type command letter
Too few command field definitions
Too many command field definitions
Trailing characters not valid in
Unexpected end-of-file
Unexpected end of line
Unknown internal function
Word not recognized
Wrong command field number
Wrong number of field type
You cannot modify this menu
```

9. THE C-SHELL

ABOUT THIS CHAPTER

This chapter serves as an introduction to the XENIX V C-Shell.

CONTENTS

INTRODUCTION

The C-shell program, **csh** , is a command language interpreter for XENIX system users. The C-shell, like the standard XENIX shell **sh** , is an interface between you and the XENIX commands and programs. It translates command lines typed at a terminal into corresponding system actions, gives you access to information, such as your login name, home directory, and mailbox, and lets you construct shell procedures for automating system tasks.

This chapter explains how to use the C-shell. It also explains the syntax and function of C-shell commands and features, and shows how to use these features to create shell procedures.

INVOKING THE C-SHELL

You can invoke the C-shell from another shell by using the **csh** command. To invoke the C-shell, type:

```
csh
```

at the standard shell's command line. You can also direct the system to invoke the C-shell for you when you log in. If you have given the C-shell as your login shell in your */etc/passwd* file entry, the system automatically starts this shell when you log in.

After the system starts the C-shell, the shell searches your home directory for the command files, *.cshrc* and *.login* . If the shell finds the files, it executes the commands contained in them, then displays the C-shell prompt.

The *cshrc* file contains the commands you wish to execute each time you start a C-shell, and the *.login* file contains the commands you wish to execute after logging in to the system. For example, the following is the contents of a typical *.login* file:

```
set ignoreeof
set mail=(/usr/spool/mail/bill)
set time=15
set history=10
mail
```

This file contains several **set** commands. The **set** command is executed directly by the C-shell; there is no corresponding XENIX program for this command. **Set** also sets the C-shell variable, *ignoreeof* , which shields the C-shell from logging out if you type **CTRL D** . Instead of **CTRL D**, you can use the **logout** command to log out of the system. By setting the *mail* variable, you tell the C-shell to watch for incoming mail and to notify you if new mail arrives.

Next the file sets the C-shell *time* variable to 15 causing the C-shell to automatically print out statistics lines for commands that execute for at least 15 seconds of CPU time. The file then sets the *history* variable to 10 indicating that the C-shell will remember the last 10 commands typed

in its *history list* .

Finally, the file invokes the XENIX *mail* program.

When the C-shell finishes processing the *.login* file, it begins reading commands from the terminal, prompting for each with:

```
%
```

When you log out (by giving the **logout** command) the C-shell prints:

```
logout
```

and executes commands from the *.logout* file if it exists in your home directory. After that, the C-shell terminates and XENIX logs you out of the system.

USING SHELL VARIABLES

The C-shell maintains a set of variables. For example, in the previous discussion, the variables, *history* and *time* , had the values 10 and 15. The values of each C-shell variable have arrays of zero or more strings. C-shell variables may be assigned values by the **set** command, which has several forms, the most useful of which is:

```
set name = value
```

Through a substitution feature, you can use C-shell variables to store values that you can use later in commands. The C-shell variables most commonly referenced are, however, those that the C-shell itself refers to. By changing the values of these variables you can directly affect the behavior of the C-shell.

One of the most important variables is PATH . This variable contains a list of directory names. When you type a command name at your terminal, the C-shell examines each named directory in turn, until it finds an executable file whose name corresponds to the name you typed. The **set** command with no arguments displays the values of all variables currently defined in the C-shell. The following example shows some typical default values:

```
argv    ()
HOME     /usr/bill
PATH     (. /bin /usr/bin)
prompt  %
SHELL    /bin/csh
status  0
```

This output indicates that the PATH variable begins with the current directory indicated by dot (.), then */bin* , and */usr/bin* . Your own local commands may be in the current directory. Normal XENIX commands reside in */bin* and */usr/bin* .

Sometimes a number of locally developed programs reside in the directory, */usr/local* . If you want all C-shells that you invoke to have access to these new programs, place the command:

```
set PATH=(. /bin /usr/bin /usr/local)
```

in the *.cshrc* file in your home directory. Try this, log out, and then log back in. Type:

```
set
```

to see that the value assigned to PATH has changed.

When you log in the C-shell examines each directory that you insert into your path and determines which commands are contained there, except for the current directory which the C-shell treats specially, This means that if commands are added to a directory in your search path after you have started the C-shell, they will not necessarily be found. If you wish to use a command which has been added after you have logged in, you can give the following command to the C-shell:

```
rehash
```

rehash causes the shell to recompute its internal table of command locations, so that it will find the newly added command. Since the C-shell has to look in the current directory for each command anyway, placing it at the end of the path specification usually works best and reduces overhead.

Other useful built in variables are: **HOME** , which shows your home directory, and **ignoreeof** , which can be set in your *.login* file to tell the C-shell not to exit when it receives an end-of-file from a terminal. The **ignoreeof** variable is one of several variables whose value the C-shell does not care about; the C-shell is only concerned with whether these variables are set or unset. Thus, to set **ignoreeof** , you simply type:

```
set ignoreeof
```

and to unset it type:

```
unset ignoreeof
```

Some other useful built-in C-shell variables are **NOCLOBBER** and **MAIL** . The syntax:

```
>filename
```

which redirects the standard output of a command just as in the regular shell, and overwrites and destroys the previous contents of the named file. In this way, you may accidentally overwrite a valuable file. If you prefer that the C-shell not overwrite files you can type:

```
set noclobber
```

in your *.login* file. Then if you type:

```
date > now
```

you will get an error message if the *now* file already exists. You can type:

```
date >! now
```

if you really want to overwrite the contents of *now* . The ''>!'' is a special syntax indicating that overwriting or ''clobbering'' the file is ok. (The space between the exclamation point (!) and the word ''now'' is critical here, since ''!now'' would be an invocation of the history feature and would have a totally different effect.)

USING THE C-SHELL HISTORY LIST

The C-shell can maintain a history list into which it places the text of previous commands. You can use metacharacter notations that represent previously used shell commands, or words from commands, to form new commands. The history feature can be used to repeat previous commands or to correct minor typing mistakes in commands.

This section provides a tutorial example to teach you how to use the history feature. In this example, you will use the history feature to create a sample file, compile the file, execute the file, and perform various other commands on the file.

To learn how to use the history feature, do the following:

1. Create a file named *bug.c* using the computer editor of your choice. The file must contain the five lines shown below:

```
main()
{
        printf("hello);
}
```

   This file is a very simple C program that has a few intentional bugs in it.

2. Display the *bug.c* file by typing the command line:

```
cat bug.c
```

   The output of this command will be:

```
main()
{
        printf("hello);
}
```

3. Compile the *bug.c* file by typing the command line:

```
cc !$
```

This command line makes use of the history feature. The exclamation mark (!) is the metacharacter used to invoke the history feature. The dollar sign ($) means to use the last argument to the previous command as the argument to this command. The output of this command line will look like:

```
cc bug.c
bug.c
bug.c(4) : warning : newline in string constant
bug.c(5) : syntax error: '}'
```

The C-shell echoes the command as if it had been typed without the use of the history feature, and then executes the command. The compilation yields error diagnostics, so now you must edit the *bug.c* file.

4. Using the computer editor of your choice, edit line 4 of the *bug.c* file to look like:

```
printf("hello");
```

5. Recompile the *bug.c* file using the command line:

```
!c
```

This command line invokes the history feature and repeats the last executed command that started with a ``c''. The output of this command line will look like:

```
cc bug.c
bug.c
```

If there were other commands beginning with the letter ``c'' executed recently, you could have typed ``!c:p'' as the command line. This command line will print the last command starting with ``c'' without executing it, so that you can check to see whether you really want to execute the implied command.

6. Run the default output file *a.out* that resulted from compiling the *bug.c* file. To do this, type the command:

```
a.out
```

The output of this command will look like:

```
hello%
```

You will note that the C-shell prompt (%) appears on the same line as the output of *a.out* . For purposes of this example, you will assume that this is a bug that you must correct.

7. Using the computer editor of your choice, edit line 4 of the *bug.c* file to look like:

```
printf("hello0);
```

This will cause the C-shell prompt to be placed on a new line when the output file is run again.

8. Recompile the *bug.c* file by typing the command line:

```
!c -o bug
```

This command line invokes the history feature, executes the last command beginning with ``c'', and uses the **-o** option to tell the compiler to name the output file *bug* rather than the default *a.out* . The output of this command line will look like:

```
cc bug.c -o bug
bug.c
```

9. Compare the sizes of the binary program images of the *a.out* and *bug* files by typing the command line:

```
size a.out bug
```

The output of this command will be similar to:

```
a.out: 4226 + 490 + 1064 = 5780 = 0x1694
bug: 4226 + 492 + 1064 = 5782 = 0x1696
```

10. List the output files *a.out* and *bug* in long format by typing the command line:

```
ls -l !*
```

This command line invokes the **ls** command, invokes the history feature, and uses all the arguments specified in the previous command as arguments to this command. The output of this command line will be similar to:

```
ls -l a.out bug
-rwxr-xr-x      1 bill  group   3932    Dec 19 09:41    a.out
-rwxr-xr-x      1 bill  group   3932    Dec 19 09:41    bug
```

11. Run the output file *bug* to verify that its output is correct. To do this, type the command:

```
bug
```

Its output will be:

```
hello
```

12. Print a program listing of the file *bug.c* by typing the command line:

```
pr bug.c | lpt
```

This command line invokes the **pr** command and pipes its output to a

lineprinter (lpt). The lineprinter notation *lpt* should really be *lpr*. We introduced an intentional spelling error in this command line so you could learn about some more neat history features. The output of this command line will be:

```
lpt: Command not found.
```

13. Correct the spelling error in the **pr** command line and request a printout of the program listing again. To do this, type the command line:

```
^lpt^lpr
```

This command line replaces the string after the first caret with the string after the second caret in the previous command, and repeats the previous command with the new string. The output of this command line will be:

```
pr bug.c | lpr
```

There are other features available for repeating commands. The **history** command prints out a numbered list of previous commands. You can also then refer to these commands by number. You can also refer to a previous command by searching for a string which appeared in it.

USING ALIASES

The C-shell has an **alias** command that can transform commands immediately after they are input. You can use this to simplify the commands that you type, to supply default arguments to commands, or to perform transformations to commands and their arguments. The **alias** command is similar to a macro facility. You can obtain some of the **alias** features by using C-shell command files. But these are in another instance of the C-shell, so they cannot directly affect the current C-shell's environment or involve commands such as **cd** , which must be done in the current C-shell.

For example, suppose there is a new version of the mail program on the system called *newmail* that you wish to use instead of the standard mail program, *mail* . If you place the C-shell command:

```
alias mail newmail
```

in your *.cshrc* file, the C-shell transforms an input line of the form:

```
mail bill
```

into a call on *newmail* . Suppose you want the **ls** command to show sizes of files, that is, to use the -s option always. In this case, you can use the **alias** command to do:

```
alias ls ls -s
```

or:

```
alias dir ls -s
```

creating a new command named **dir** . If you then type:

```
dir ~bill
```

the C-shell translates this to:

```
ls -s /usr/bill
```

Note that the tilde (~) is a special C-shell symbol that represents the user's home directory.

Thus, you can use the **alias** command to provide short names for commands, to provide default arguments, and to define new short commands in terms of other commands. You can also define aliases with multiple commands or pipelines, showing where the arguments to the original command are to be substituted by using the history feature. Thus, the following definition:

```
alias cd 'cd * ; ls '
```

specifies an **ls** command after each **cd** command. The entire alias definition is enclosed in single quotation marks (' ') to prevent most substitutions from occurring and to prevent the semicolon (;) from being recognized as a metacharacter. The exclamation mark (!) is escaped with a backslash (\) to prevent it from using its standard interpretation when the **alias** command is used. The ''\!*'' substitutes the entire argument list to the **cd** command; no error is given if there are no arguments. The semicolon, which separates commands, indicates that the commands are to be done in sequence. Similarly, the following example defines a command that looks up its first argument in the password file:

```
alias whois 'grep ^ /etc/passwd'
```

The C-shell reads the *.cshrc* file each time it starts up. If you place a large number of aliases there, the C-shell will start slowly. Try to limit your aliases to a reasonable number (10 or 15). Too many aliases cause delays and make the system seem sluggish when you execute commands from within an editor or from within other programs.

REDIRECTING INPUT AND OUTPUT

Commands have a diagnostic output in addition to their standard output. This diagnostic output is normally directed to the terminal even if the standard output is redirected to a file or a pipe. Occasionally, it is useful to direct the diagnostic output along with the standard output. For instance, if you want to redirect the output of a long running command into a file and wish to have a record of any error diagnostic it produces, you can use a command line with the following form:

command >& *file*

The ''>&'' tells the C-shell to route the diagnostic output and the

standard output into *file* . Similarly, you can give the command:

command |& lpr

to route standard and diagnostic output through the pipe to the lineprinter. You can also use command line of the form:

command >&! *file*

if the *file* already exists and cannot be overwritten (``noclobber'' is set).

Finally, you can use the form:

command >> *file*

to append output to the end of an existing file. If ''noclobber'' is set, an error results if *file* does not exist, otherwise the C-shell creates *file* . The command line form:

command >>! *file*

allows you to append a file even if it does not exist and ''noclobber'' is set.

CREATING BACKGROUND AND FOREGROUND JOBS

When you type one or more commands together as a pipeline or as a sequence of commands separated by semicolons, the C-shell creates a single job consisting of these commands taken as a unit. Single commands without pipes or semicolons create the simplest jobs. Usually, every line that you type to the C-shell creates a job. For example, each of the following lines creates a job:

```
sort < data
ls -s | sort -n | head -5
mail harold
```

If you type the ampersand (&) metacharacter at the end of the commands, you start the job as a background job. So the C-shell does not wait for the job to finish, but immediately prompts for another command. The job runs in the background at the same time that normal jobs, called foreground jobs, continue to be read and executed by the C-shell. Thus, you can run the **du** program by typing:

```
du > usage &
```

which reports on the disk usage of your working directory, puts the output into the *usage* file, and returns immediately with a prompt for the next command without waiting for **du** to finish. The **du** program continues executing in the background until it finishes, even though you can type and execute more commands in the mean time. Background jobs are unaffected by signals from the keyboard such as the **INTERRUPT** or **QUIT** signals.

The **kill** command terminates a background job immediately. Normally, you do this by specifying the process number of the job you want killed. You can list the process numbers by using the **ps** command.

USING BUILT-IN COMMANDS

This section explains how to use some of the built-in C-shell commands.

The **alias** command previously described is used to assign new aliases and to display existing aliases. With no arguments, **alias** prints the list of current aliases. You may also give one argument, such as to show the current alias for a given string of characters. For example:

```
alias ls
```

prints the current alias for the string, ''ls''.

The **history** command displays the contents of the history list. You can use the numbers given with the history events to reference previous events that are difficult to reference contextually. There is also a C-shell variable named *prompt* . By using an exclamation point (!) as the value of *prompt* , the C-shell substitutes the number of the current command in the history list. You can use this number to refer to a command in a history substitution. For example, you could type:

```
set prompt=' % '
```

Note that the exclamation mark (!) had to be escaped here even within backslashes.

The **logout** command terminates a login C-shell that has **ignoreeof** set.

The **rehash** command causes the C-shell to recompute a table of command locations. This recomputation is necessary if you add a command to a directory in the current C-shell's search path and expect the C-shell to find it. Otherwise, the hashing algorithm may tell the C-shell that the command wasn't in that directory when the hash table was computed.

The **repeat** command repeats a command several times. Thus, to make 5 copies of the *one* file in the *five* file you can type:

```
repeat 5 cat one >> five
```

The **setenv** command sets variables in the environment. Thus:

```
setenv TERM adm3a
```

sets the value of the **TERM** environment variable to "adm3a". The **env** program prints out the environment. For example, its output might look like this:

```
HOME=/usr/bill
SHELL=/bin/csh
PATH=:/usr/ucb:/bin:/usr/bin:/usr/local
TERM=adm3a
USER=bill
```

The **source** command forces the current C-shell to read commands from a file. Thus, you can use the command line:

```
source .cshrc
```

after making a change to the *.cshrc* file that you wish to take effect before the next time you login.

The **time** command causes a command to be timed no matter how much CPU time it takes. Thus, the command line:

```
time cp /etc/rc /usr/bill/rc
```

displays:

```
0.0u 0.1s 0:01 8%
```

Similarly, the command line:

```
time wc /etc/rc /usr/bill/rc
```

displays:

```
      52     178    1347 /etc/rc
      52     178    1347 /usr/bill/rc
     104     356    2694 total
0.1u 0.1s 0:00 13%
```

This display indicates that the **cp** command used a negligible amount of user time (u) and about 1/10th of a second of system time (s); the elapsed time was 1 second (0:01). The word count command, **wc** , used 0.1 seconds of user time and 0.1 seconds of system time in less than a second of elapsed time. The percentage, ''13%'', indicates that over the period when it was active the **wc** command used an average of 13 percent of the available CPU cycles of the machine.

The **unalias** and **unset** commands remove aliases and variable definitions from the C-shell.

CREATING COMMAND SCRIPTS

You can place commands in files and invoke C-shells to read and execute commands from these files, called C-shell scripts. This section describes the C-shell features that are useful when creating C-shell scripts.

USING THE ARGV VARIABLE

A **csh** command script may be interpreted by saying:

csh *script argument...*

where *script* is the name of the file containing a group of C-shell commands, and *argument* is a sequence of command arguments. The C-shell places these arguments in the **argv** variable and then begins to read commands from *script* . These parameters are accessed just as any other C-shell variables.

You can make the *script* file executable by typing:

```
chmod 755 script
```

or:

```
chmod +x script
```

Place a C-shell comment at the beginning of the C-shell script (i.e., begin the file with a number sign (#)). This will cause the */bin/csh* file to be invoked automatically to execute *script* when you type:

```
script
```

If the file does not begin with a number sign (#), the standard shell, */bin/sh* , is used to execute it.

SUBSTITUTING SHELL VARIABLES

After each input line is broken into words and history substitutions are done on it, the input line is parsed into distinct commands. Then before each command is executed, the C-shell performs variable substitution on these commands. Keyed by the dollar sign ($), this substitution replaces the names of variables by their values. Thus, the command line:

```
echo $argv
```

when placed in a command script echoes the current value of the *argv* variable to the output of the C-shell script. It is an error for *argv* to be **unset** at this point.

A number of notations are provided for accessing components and attributes of variables. The notation:

```
$? name
```

expands to 1 if *name* is **set** or to 0 if *name* is not **set** . This notation is the fundamental tool for checking whether particular variables have been assigned values. All other references to undefined variables cause errors.

The notation:

```
$# name
```

expands to the number of elements in the *name* variable. To illustrate, examine the following terminal session:

```
% set argv=(a b c)
% echo $?argv
1
% echo $#argv
3
% unset argv
% echo $?argv
0
% echo $argv
Undefined variable: argv.
%
```

You can also access the components of a variable that has several values by typing:

```
$argv[1]
```

gives the first component of *argv* , or in the previous example, **a** . Similarly:

```
$argv[$#argv]
```

would give **c** , and:

```
$argv[1-2]
```

would give:

a b

Other useful notations in C-shell scripts are:

$ *n*

where *n* is an integer. This is shorthand for:

$argv[*n* **]**

the

n th parameter, and:

```
$*
```

which is a shorthand for:

```
$argv
```

The following notation:

```
$$
```

expands to the process number of the current C-shell. Since this process number is unique in the system, you can use it in the generation of unique temporary filenames.

One minor difference between ''$ *n* '' and ''$argv[*n*]'' should be noted here. The form ''$argv[*n*]'' yields an error if *n* is not in the range, 1-$#argv, while ''$ *n* '' never yields an out-of-range subscript error. This difference is necessary for compatibility with the way previous shells handle parameters.

Another important point is that it is never an error to give a subrange of the form ''n-''; if there are less than ''n'' components of the given variable, no words are substituted. A range of the form, ''m-n'', also returns an empty vector without giving an error when ''m'' exceeds the number of elements of the given variable. An empty vector without an error returns only if the subscript ''n'' is in range.

USING EXPRESSIONS

To construct useful C-shell scripts, the C-shell evaluates expressions that are based on the values of variables. In fact, all the arithmetic operations of the C language are available in the C-shell with the same precedence that they have in C. In particular, the operations, ''=='' and ''!='', compare strings, and the operators, ''&&'' and ''||'', implement the logical AND and OR operations. The special operators, ''=~'' and ''!~'', are similar to ''=='' and ''!='' except that the right-hand string can have pattern matching characters (*, ?, or [and]). These operators test whether the string on the left matches the pattern on the right.

The C-shell also allows file inquiries of the form:

```
-? filename
```

where the question mark (?) is replaced by a number of single characters. For example, the primitive expression:

```
-e filename
```

tells whether *filename* exists. Other primitive expressions test for read, write, and execute access to the file, whether it is a directory, or whether it has nonzero length.

You can test whether a command terminates normally, by using a primitive expression of the form:

{ *command* }

which returns 1 if the command exits normally with exit status 0, or 0 if the command terminates abnormally or with exit status nonzero. If you want more detailed information about the execution status of a command, you can execute it and examine the **status** variable in the next command. Since the **$status** variable is set by every command, its value is always changing.

USING THE C-SHELL: A SAMPLE SCRIPT

The following sample C-shell script uses the expression feature of the C-shell as well as some of its control structures:

```
#
# Copyc copies those C programs in the specified list
# to the directory ~/backup if they differ from the files
# already in ~/backup
#
set noglob
foreach i ($argv)

        if ($i !~ *.c) continue  # not a .c file so do nothing

        if (! -r ~/backup/$i:t) then
                echo $i:t not in backup... not cp\'ed
                continue
        endif

        cmp -s $i ~/backup/$i:t # to set $status

        if ($status != 0) then
                echo new backup of $i
                cp $i ~/backup/$i:t
        endif
end
```

This script uses the **foreach** command, which iteratively executes the group of commands between the **foreach** and **end** statements for each valid value of the variable *i* . If you want to look more closely at what happens during execution of a **foreach** loop, you can use the debug command **break** to stop execution at any point and the debug command **continue** to resume execution. The value of the iteration variable (*i* in this case) will stay at whatever it was when the last **foreach** loop was completed.

You set the **noglob** variable to prevent filename expansion of the members of **argv** . This is a good idea if the arguments to a C-shell script are filenames which have already been expanded or if the arguments may contain metacharacters for filename expansion.

The other control construct is a statement with the form:

```
if ( expression ) then
    command
      ...
endif
```

The placement of the keywords in this statement is inflexible due to the current implementation of the C-shell. The following two formats are not acceptable to the C-shell:

```
if ( expression ) # Won't work!
then
    command
            ...
endif
```

and:

```
if ( expression ) then command endif # Won't work
```

The C-shell does have another form of the **if** statement:

```
if ( expression ) command
```

which can be written:

```
if ( expression )
    command
```

The command must not involve ''|'', ''&'' or '';'' and must not be another control command. The second form requires that the final backslash (\) immediately precedes the end-of-line. More general **if** statements also admit a sequence of **else-if** pairs followed by a single **else** and an **endif** , for example:

```
if ( expression ) then
    commands

else if ( expression ) then
    commands
    ...
else
    commands
endif
```

Another important feature in C-shell scripts is the colon (:) modifier. You can use the :r modifier to extract the root of a filename, or you can use :e to extract the extension. Thus, if the i variable has the value */mnt/foo.bar*

```
echo $i $i:r
```

produces:

```
/mnt/foo.bar /mnt/foo
```

This example shows how the :r modifier strips off the trailing ''.bar''. Other modifiers take off the last component of a pathname leaving the :h head or all but the last component of a pathname leaving the tail, :t . You can also use the command substitution feature to modify strings. Since each usage of this feature involves the creation of a new process, it is more expensive than the colon (:) modification feature. Also, note that the current implementation of the C-shell limits the number of colon modifiers on a $ substitution to 1. Thus:

```
% echo $i $i:h:t
```

produces:

```
/a/b/c /a/b:t
```

Finally, note that the number sign character (#) lexically introduces a C-shell comment in C-shell scripts (but not from the terminal). All subsequent characters on the input line after a # are discarded by the C-shell. You can put this # character in quotation marks, using ''#'' or ''\#'' to place it in an argument word.

USING OTHER CONTROL STRUCTURES

The C-shell also has the control structures, **while** and **switch** , which are similar to those control structures of C. These take the following forms:

```
while ( expression )
    commands
end
```

and:

```
switch ( word )

case strl:
    commands

    breaksw
    .
    .
    .
    case strn:

        commands
        breaksw

    default:
        commands
        breaksw
    endsw
```

C programmers should note that **breaksw** exits from a **switch** , while **break** exits a **while** or **foreach** loop. The two commands are often confused.

Finally, the C-shell allows a **goto** statement, with labels that resemble C labels:

```
loop:
    command
    .
    .
    .
    command
goto loop
```

SUPPLYING INPUT TO COMMANDS

Commands run from C-shell scripts receive by default the standard input of the C-shell which is running the script. It allows C-shell scripts to fully participate in pipelines, but mandates extra notation for commands that are to take inline data. Thus we need a metacharacter notation for supplying inline data to commands in C-shell scripts. For example, consider the following script which uses the editor to delete leading blanks from the lines in each specified argument file:

```
# deblank -- remove leading blanks
foreach i ($argv)
ed - $i << ' EOF'
1,$s/^[ ]*//
w
q
'EOF'
end
```

The notation, ''<< 'EOF''', means that the standard input for the **ed** command is to come from the text in the C-shell script file up to the next line consisting of exactly **EOF** . The fact that the **EOF** is enclosed in single quotation marks ('), causes the C-shell not to perform variable substitution on the intervening lines. In this case, since the ''1,$'' form was used in the editor script, you need to ensure that this dollar sign is not a variable substitution. You can also make sure of this by typing a backslash (\), before the dollar sign ($), as shown in the following example:

```
1,\$s/^[ ]*//
```

Quoting the EOF terminator is a more reliable way of verifying that the dollar sign ($) is not a variable substitution.

CATCHING INTERRUPTS

If your C-shell script creates temporary files, you may want to catch interruptions of the C-shell script so that you can clean up these files. You can start this process by issuing a command line of the following form:

```
onintr label
```

where *label* is a label in your program. If the C-shell receives an interrupt, it does a ''goto label'' allowing you to remove the temporary files. The C-shell then does an **exit** command to exit from the C-shell script. If you wish to exit with nonzero status you can write:

```
exit (1)
```

to exit with status 1.

USING OTHER FEATURES

There are other features of the C-shell that are useful for writing procedures. You can use the **verbose** and **echo** options and the related -v and -x command line options to trace the actions of the C-shell. The -n option causes the C-shell to read commands and not to execute them, which can be useful.

Also note that the C-shell will not execute C-shell scripts that do not begin with the number sign character (#). In other words, you cannot execute C-shell scripts that begin with comments.

Another quotation feature is the double quotation mark ("), which allows only some expansion features to occur on the quoted string. This feature also makes this string into a single word as the single quotation mark (') does.

STARTING A LOOP AT A TERMINAL

You might also use the **foreach** control structure at the terminal to perform a number of similar commands. For instance, if there are three shells in use on a particular system, */bin/sh* , */bin/nsh* , and */bin/csh* , you can count the number of persons using each shell by using the following commands:

```
grep -c csh$ /etc/passwd
grep -c nsh$ /etc/passwd
grep -c -v sh$ /etc/passwd
```

Since these commands are similar, you can use the **foreach** command to simplify them:

```
$ foreach i ('sh$' 'csh$' '-v sh$')
? grep -c $i /etc/passwd
? end
```

Note that the C-shell prompts for input with ''?'' when reading the body of the loop. This occurs only when you enter the **foreach** command interactively.

Also useful with loops are variables that contain lists of filenames or other words are useful in loops. For example, note the following terminal session:

```
% set a=(`ls`)
% echo $a
csh.n csh.rm
% ls
csh.n
csh.rm
% echo $#a
2
```

In the previous example, the **set** command assigns the *a* variable a list of all the filenames in the current directory. You can then iterate over these names to perform any chosen function.

The C-shell converts the output of a command within back quotation marks (` `) to a list of words. You can also place the quoted string within double quotation marks (" ") to take each (nonempty) line as a component of the variable. This action prevents the lines from being split into words at spaces and tabs. An :x modifier exists which you can use later to expand each component of the original variable into another variable by splitting it into separate words at embedded spaces and tabs.

USING BRACES WITH ARGUMENTS

Another form of filename expansion uses the characters, ''{'' and ''}''. These characters specify that the contained strings, separated by commas (,) are to be consecutively substituted into the containing characters with the results expanded left to right. Thus:

```
A{str1, str2, ... strn}B
```

expands to:

```
Astr1B Astr2B ... AstrnB
```

This expansion occurs before the other filename expansions, and may be applied recursively (i.e., nested). The results of each expanded string are sorted separately, preserving left to right order. The resulting filenames are not required if no other expansion features are used. This means that you can use this feature to generate arguments which are not filenames, but which have common parts.

A typical example of this is:

```
mkdir ~/{hdrs,retrofit,csh}
```

which you can use to create the subdirectories, *hdrs* , *retrofit*, and *csh*,

in your home directory. This feature is useful when the common prefix is longer than in this example, such as:

```
chown root /usr/demo/{file1,file2,...}
```

SUBSTITUTING COMMANDS

Before filenames are expanded, any command enclosed in accent symbols (` `) is replaced by the output from that command. Thus, you can use the following command line:

```
set pwd=`pwd`
```

to save the current directory in the *pwd* variable or to type:

```
vi `grep -l TRACE *.c`
```

to run the **vi** editor, supplying as arguments those files ending in *.c* which have the ''TRACE'' string in them. Command expansion also occurs to input that is redirected with ''<<'' and within double quotation marks (" ").

SPECIAL CHARACTERS

The following table lists the special characters used by **csh** and the XENIX system. A number of these characters also have special meaning in expressions.

Syntactic metacharacters

; Separates commands to be executed sequentially

| Separates commands in a pipeline

( ) Brackets expressions and variable values

& Follows commands to be executed without waiting for completion

Filename metacharacters

/ Separates components of a file's pathname

. Separates root parts of a filename from extensions

? Expansion character matching any single character

* Expansion character matching any sequence of characters

[ ] Expansion sequence matching any single character from a set of characters

~ Used at the beginning of a filename to indicate home directories

{ } Used to specify groups of arguments with common parts

Quotation metacharacters

\ Prevents meta-meaning of following single character

' Prevents meta-meaning of a group of characters

" Same as ', but allows variable and command expansion

Input/output metacharacters

< Indicates redirected input

> Indicates redirected output

Expansion/Substitution Metacharacters

$ Indicates variable substitution

! Indicates history substitution

: Precedes substitution modifiers

^ Used in special forms of history substitution

` Indicates command substitution

Other Metacharacters

Begins scratch filenames; indicates C-shell comments

- Prefixes option (flag) arguments to commands

10. USING vi

ABOUT THIS CHAPTER

This chapter serves as an introduction to the XENIX V text editor, vi .

CONTENTS

INTRODUCTION

The XENIX system provides two text editors, **ed** and **vi**, that let you create and modify any ASCII text file. This chapter discusses the use of **vi** and is organized as follows:

SEE THE SECTION CALLED	FOR...
Demonstration	a hands-on demonstration of basic **vi** concepts
Editing Tasks	reference to specific editing tasks
Solving Common Problems	problem solving information
Setting up your Environment	information on setting up your **vi** environment
Command Summary	a summary of **vi** commands

Because **vi** is such a powerful editor, it has many more commands than you can learn at one sitting. If you have not used a text editor before, the best approach is to become thoroughly comfortable with the concepts and operations presented in the demonstration section, then refer to the second part for specific tasks you need to perform. Once you are familiar with the basic **vi** commands, you can easily learn to use the more advanced features.

If you have used a text editor before, you may want to turn directly to the task-oriented part of this chapter. Begin by learning the features you will use most often.

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

DEMONSTRATION

The following demonstration gives you hands-on experience using **vi** , and introduces some basic concepts that you must understand before you can learn more advanced features. This demonstration should take one hour. Remember that the best way to learn **vi** is to actually use it, so don't be afraid to experiment.

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

Note

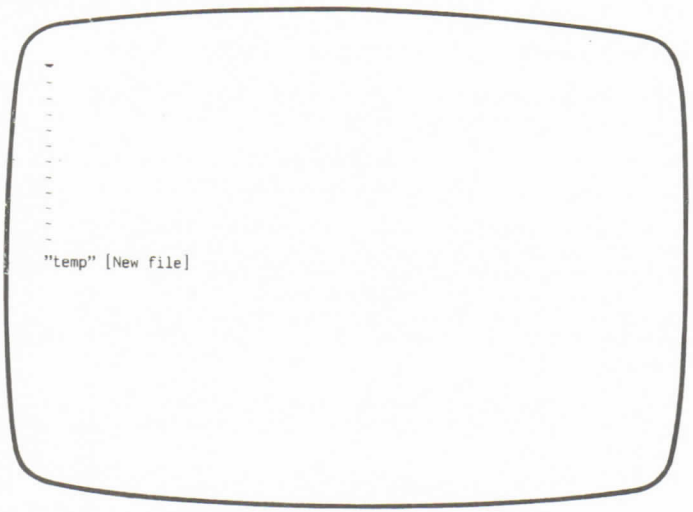
Throughout this chapter, the character, word or line affected by the cursor is referred to as being "current": e.g., the current line is the line on which the cursor resides.

ENTERING THE EDITOR

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

```
vi temp
```

Your screen will look like this:



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

The top line of your display is the only line in the file and is marked by the cursor, shown as an underline character; tildes indicate lines on the screen only, not real lines in the file.

INSERTING TEXT

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

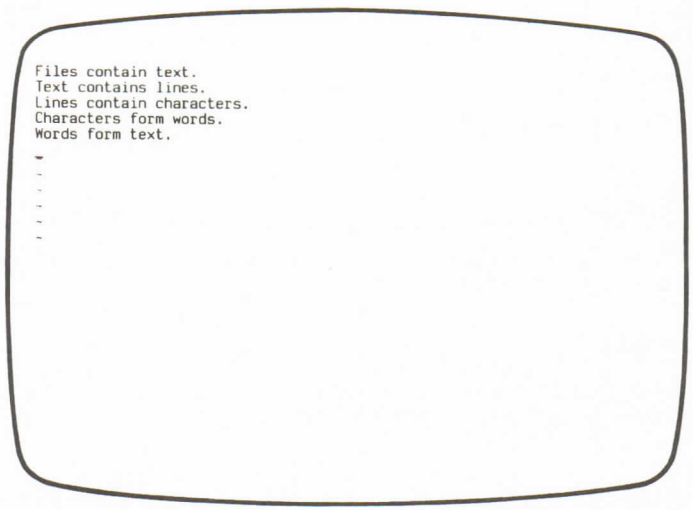
```
i
```

Like most vi commands, the **i** command doesn't appear on your screen. The command itself switches you from command mode to insert mode:

- In insert mode every character you type appears on the screen.
- In command mode XENIX interprets the characters you type as commands: the characters don't appear on screen.

If you are not certain which mode you are in, press until you hear the bell: this indicates that you are in command mode.

Next, type the following five lines. Press **CR** at the end of each line. If you make a mistake, use the **BKSP** key to erase the error.

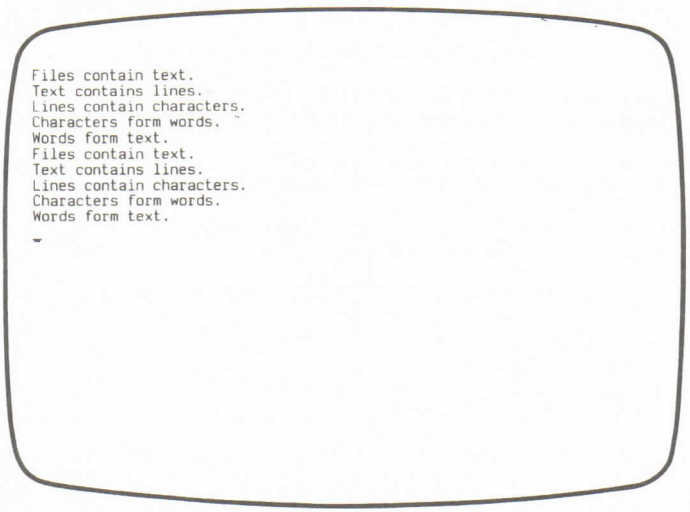


Press **ESC** when you are finished.

REPEATING A COMMAND

The repeat command repeats the most recent insertion or deletion. To execute the repeat command, press the period (.).

So, to add five more lines of text, enter command mode, and press .; your screen will look like this:



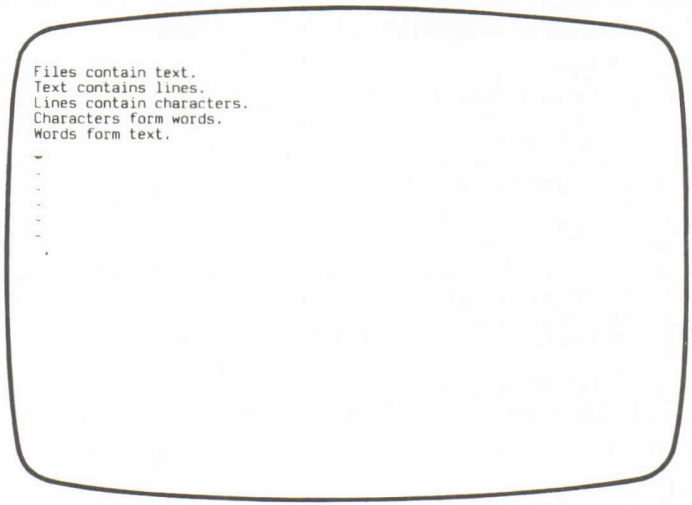
The repeat command is repeated relative to the location of the cursor and inserts text below the current line.

UNDOING A COMMAND

Another useful command is the **undo** command, **u**. **undo** lets you recover from inadvertent deletions or insertions. Type:

```
u
```

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



Now type:

u

again, and the five lines are reinserted.

MOVING THE CURSOR

In addition to the arrow keys, the following letter keys let you move the cursor around the screen. Since you're still in command mode, give them a try.

TYPE	TO MOVE THE CURSOR...
h	left
l	right
k	up
j	down

Remember that the cursor movement keys work only in command mode. You may want to try the following additional cursor movement commands:

TYPE	TO MOVE THE CURSOR...
w	forward one word.
b	back one word.
0	to the beginning of a line.
$	to the end of a line.
n G	to the specified line in the file. G alone places the cursor on the last line in the file.
H	to the upper-left corner of the screen.
L	to the bottom line on the screen.

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

PRESS	TO SCROLL...
CTRL U	up 1/2 screen.
CTRL D	down 1/2 screen.
CTRL F	forward one screenful.
CTRL B	back one screenful.

DELETING

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

dd Deletes the current line, regardless of the location of the cursor in the line.

dw Deletes the current word. If the cursor is in the middle of the word, deletes from the cursor to the end of the word.

x Deletes the current character.

d$ Deletes from the cursor to the end of the line.

D Deletes from the cursor to the end of the line.

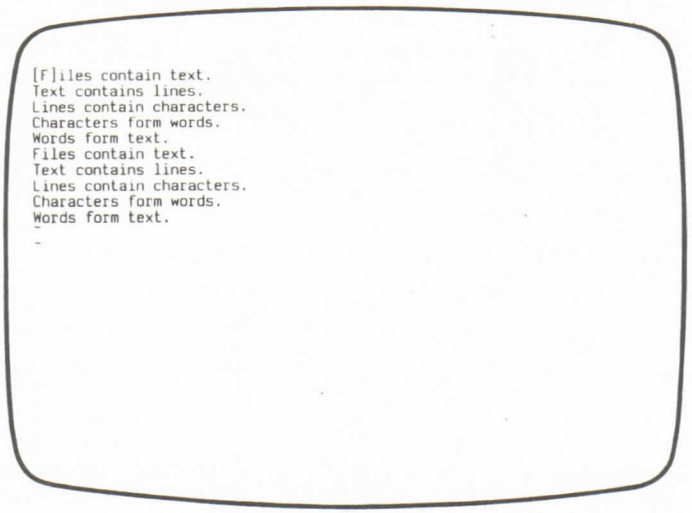
d0 Deletes from the cursor to the beginning of the line.

To learn how these commands work, we'll delete various parts of the demonstration file.

1. Press **ESC** to enter command mode.
2. Move to the first line of the file by typing:

   1G

   At first, your file should look like this:

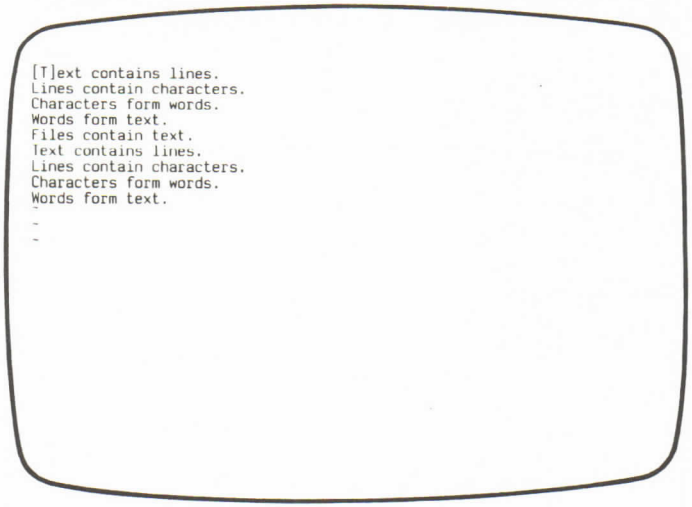


Note Throughout this chapter the current character will be bracketed ([]).

3. Delete the current line by typing:

   dd

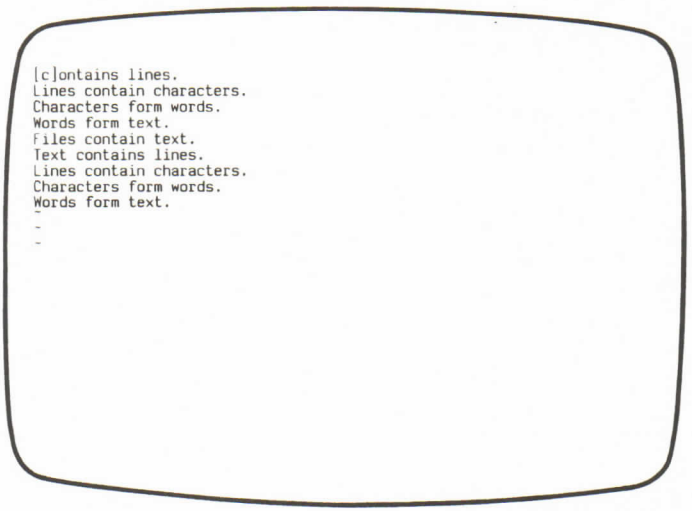
   Your file should now look like this:



4. Delete the current word by typing:

   dw

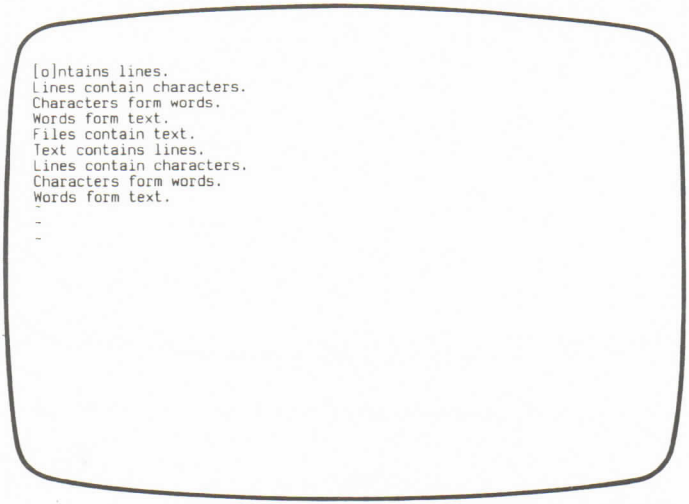
   Your file should look like this:



5. Delete the current character above the cursor by typing:

```
x
```

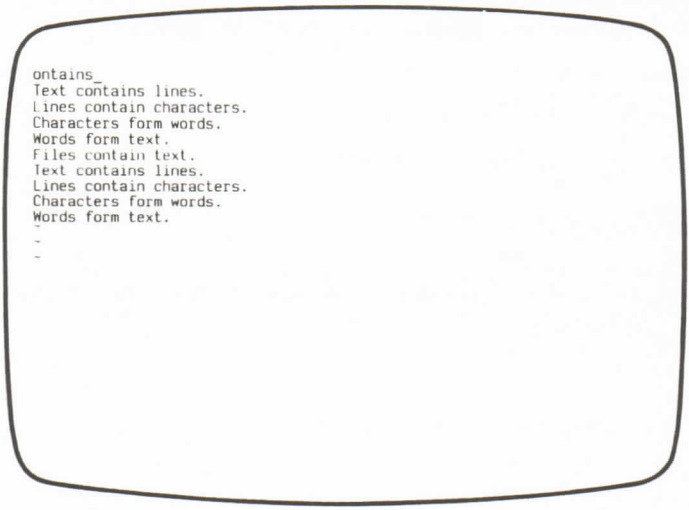
Your file should look like this:



6. Type a **w** command to move your cursor to the beginning of the word "lines" on the first line. Delete to the end of the line by typing:

```
d$
```

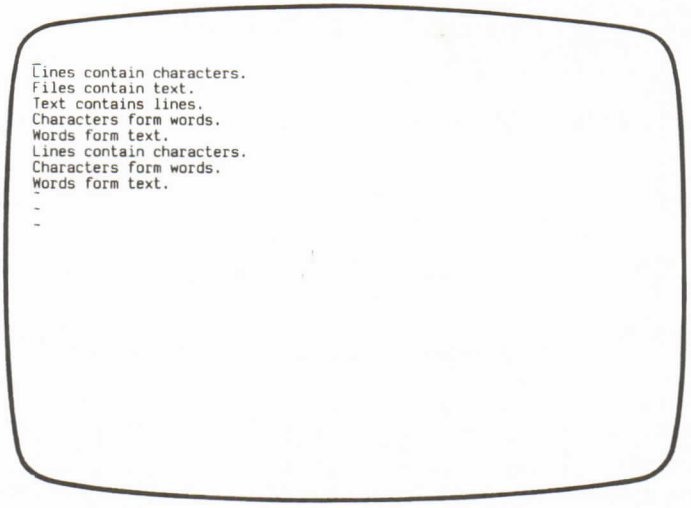
Your file looks like this:



7. Delete all the characters on the line before the cursor by typing:

   d0

   This leaves a single space on the line:



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

8. Type **i** to enter insert mode, then type:

```
Files contain text.
Text contains lines.
```

9. Press **ESC** to return to command mode.

SEARCHING FOR A PATTERN

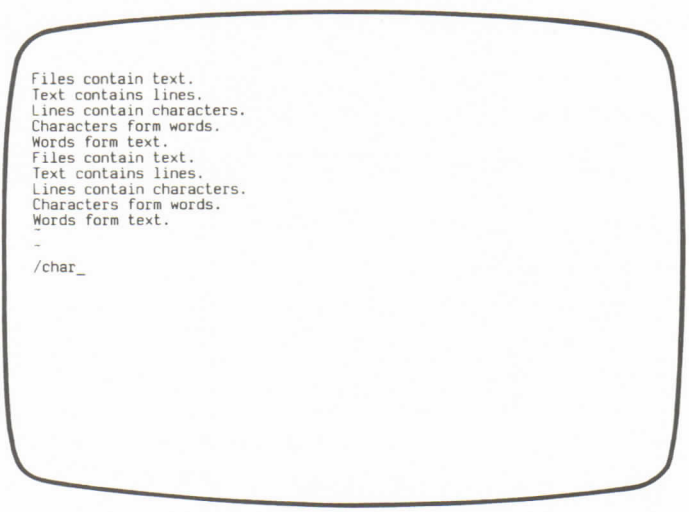
You can search forward in a file for a pattern of characters by typing a slash (/) followed by the pattern you are searching for, terminated by a **CR**. For example, make sure you are in command mode (press **ESC**) then type:

H

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

/char

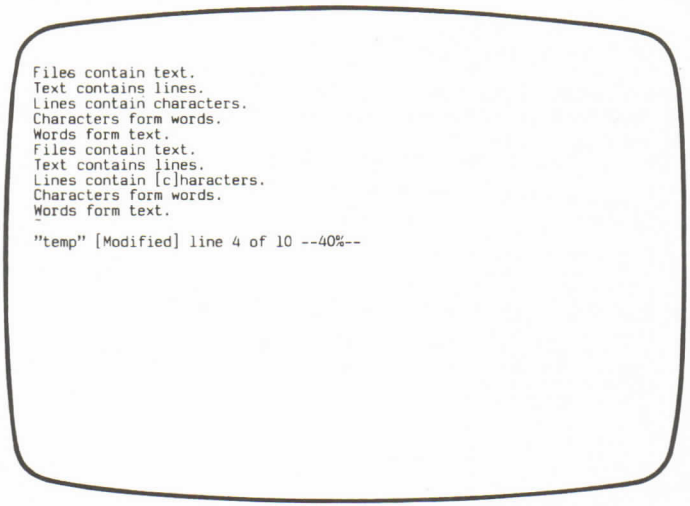
Don't press **CR** yet. Your screen should look like this:



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

Note that the slash character and the pattern that you are searching for appear at the bottom of the screen. This bottom line is the vi status line. The status line displays information, including patterns you are searching for, line-oriented commands (explained later in this demonstration), and error messages.

For example, to get status information about the file, press **CTRL G** . Your screen should look like this:



The status line tells you:

- the name of the file
- whether it has been modified
- the current line number
- the number of lines in the file
- your location in the file as a percentage of the number of lines in the file

The status line disappears as you continue working.

SEARCHING AND REPLACING

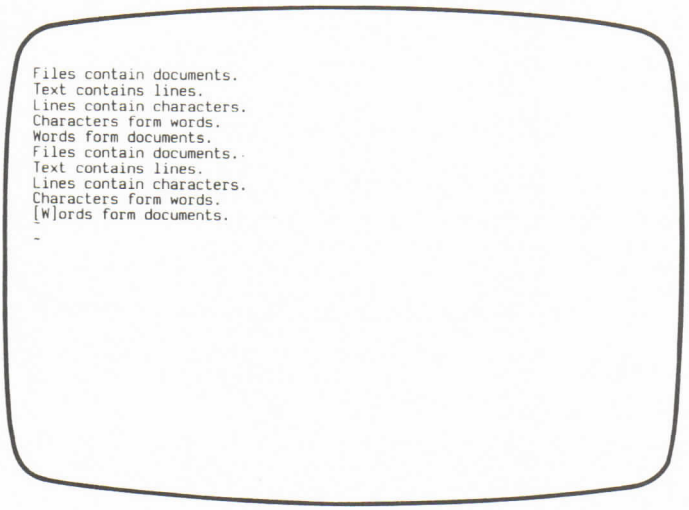
The XENIX search and replace feature lets you use one command to change all occurrences of a text string. The command will appear on the status line as you type it. For example, to change "text" to "documents":

1. Press **ESC** to enter command mode.

2. Type:

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

This command means "From the first line (1) to the end of the file ($), find text and replace it with documents (s/text/documents/) everywhere it occurs on each line (g)."
Press **CR** . Your screen should look like this:

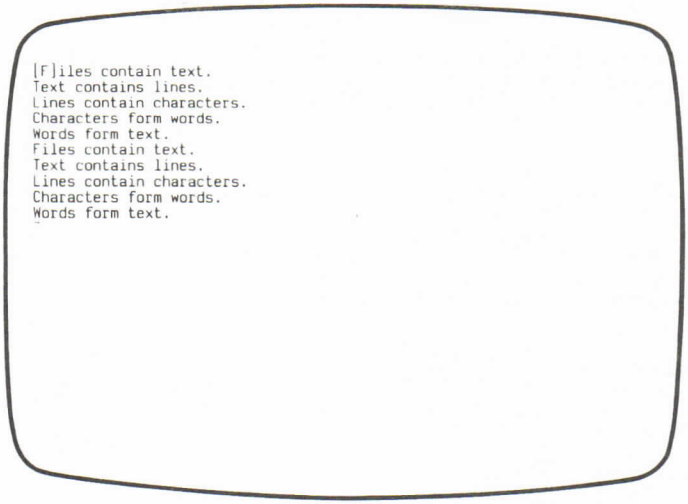


Note that "Text" in lines two and eight was not changed. Case is significant in searches.

Just for practice, use the **undo** command to change "documents" back to "text". Type:

```
u
```

Your screen now looks like this:



The commands you have learned so far have all been screen-oriented: screen-oriented commands execute at the location of the cursor. You do not need to tell the computer where to perform the operation; it takes place relative to the cursor. Commands that can perform more than one action (searching and replacing) are line-oriented commands: line-oriented commands require you to specify an exact location (called an "address") where the operation is to take place. Screen-oriented commands are easy to type in, and provide immediate feedback; the change is displayed on the screen. Line-oriented commands are more complicated to type in, but they can be executed independent of the cursor, and in more than one place in a file at a time.

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

Note

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

LEAVING vi

All of your editing has affected a copy of the file temp, that you specified when you invoked **vi**. To save these changes, exit the editor and return to the XENIX shell, by typing:

```
:x
```

The name of the file and the number of lines and characters it contains appear on the status line:

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

The XENIX prompt will then appear below the status line.

ADDING TEXT FROM ANOTHER FILE

In this section we'll create a new file, and insert text from another file into it.

1. Create a new file named practice:

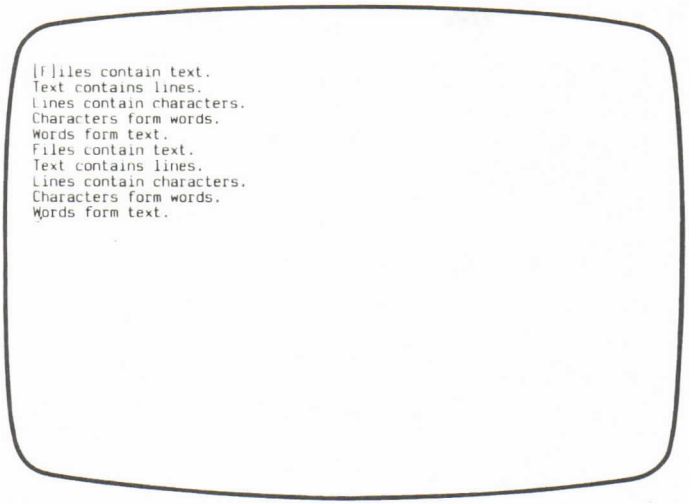
   ```
   vi practice
   ```

2. Press **ESC** to enter command mode.

3. Use the line-oriented read command to copy the text from temp to practice:

   ```
   :r temp
   ```

Your file should look like this:



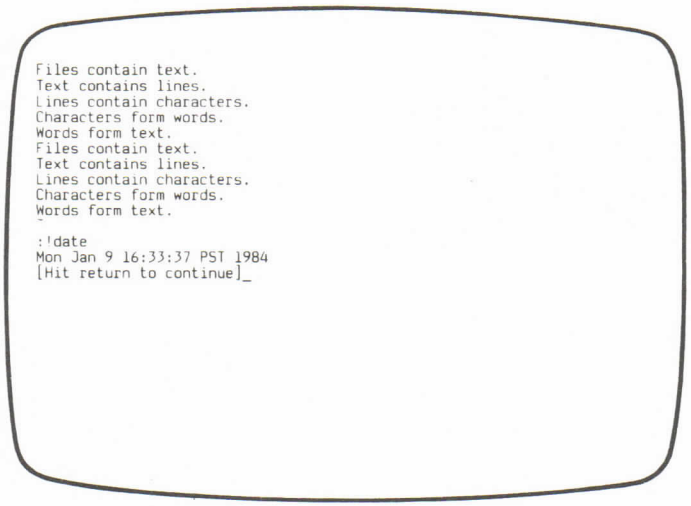
There is an empty line at the top of the file. Move the cursor to the empty line and delete it with the **dd** command.

LEAVING VI TEMPORARILY

You can exit **vi** temporarily to execute commands by preceding the command with an exclamation point on the status line. For example, to find out the date and time, type:

```
:!date
```

This displays the date, then prompts you to press **CR** to reenter command mode. Your screen should look similar to this:



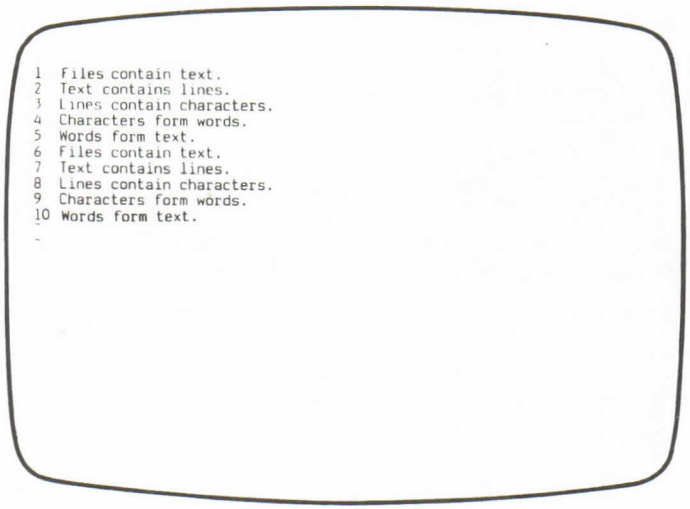
TURNING ON LINE NUMBERING

vi lets you control a number of editing parameters such as line number display.

To turn on automatic line numbering, type:

```
:set number
```

XENIX redraws your screen, and places line numbers to the left of the text:



To see the complete list of available options, type:

```
:set all
```

See "Setting up your Environment" later in this chapter for information on setting these options.

EXITING THE EDITOR

To exit vi , type:

```
:q
```

This completes the demonstration. You have learned the fundamentals of vi. The following sections provide more detailed information about these fundamentals and about vi other commands and features.

EDITING TASKS

This section explains how to perform common editing tasks.

ENTERING THE EDITOR

There are several ways to begin editing, depending on what you are planning to do. The most common way to enter **vi** is to type "vi" and the name of the file you wish to edit:

vi *filename*

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

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

vi +100 *filename*

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

To begin editing at the first occurrence of a particular word, type:

vi +/ *word filename*

The cursor is placed at the first occurrence of *word* .

See "Editing a Series of Files" for how to invoke **vi** on more than one file at a time.

MOVING THE CURSOR

Cursor movement keys let you move the cursor within a file. You must be in command mode to move the cursor.

Moving the Cursor By Character

PRESS			TO MOVE THE CURSOR...
SPACEBAR	or	**l**	forward
BACKSPACE	or	**h**	backward
k			up
j			down

The cursor moves one character, unless you specify a number of characters. For example, to move backward four characters, type:

4h

To move the cursor to a designated character on the current line, use the **f** key:

PRESS	TO MOVE THE CURSOR...
F	backward
f	forward

For example, to move the cursor backward to the nearest "p" on the current line, type:

```
Fp
```

To move the cursor forward to the nearest "p", type:

```
fp
```

The **T** and **t** keys work like **f** and **F**, but place the cursor immediately after the specified character. For example, to move the cursor back to the space next to the nearest "p" in the current line, type:

```
Tp
```

If the "p" were in the word telephone, the cursor would be positioned on the "n".

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

Moving the Cursor by Words

The **w** key moves the cursor forward to the beginning of the specified number of words. Punctuation and nonalphabetic characters are considered words, and will be counted as such. For example, if the cursor rests on the first letter of the sentence:

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

and you press:

```
6w
```

the cursor stops on the "k" in know. The comma and apostrophe are considered words.

W works like **w**, but includes punctuation and nonalphabetic

characters as part of the word. Using the previous example, if you press:

```
6W
```

the cursor stops on the "r" in "returned"; the comma and the apostrophe are considered part of the words they punctuate.

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

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

```
4b
```

and the cursor moves to the "t" in didn't. Type:

```
4B
```

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

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

Moving the Cursor by Lines

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

```
6+
```

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

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

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

```
4$
```

CTRL P and **k** move the cursor backward a specified number of lines, keeping it on the same place on the line. For example, to move the cursor backward four lines from the current line, type:

```
4k
```

Moving to Main Screen Locations

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

MOVING AROUND IN A FILE: SCROLLING

The following commands let you display different portions of the file on screen:

PRESS	TO SCROLL...
CTRL U	up one-half screen
CTRL B	up one full screen
CTRL D	down one-half screen
CTRL F	down one full screen

Placing a Line at the Top of the Screen

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

```
z
```

To place a specific line at the top of the screen, precede z with the line number, as in:

```
33z
```

For information on how to display line numbers, see "Displaying Line Numbers: number".

INSERTING TEXT

You can insert text anywhere on a line. In order to insert text into a file, you must be in insert mode. To enter insert mode, press one of the following keys:

PRESS	TO ENTER INSERT MODE...
i	at the current cursor location (text inserted before the cursor)
I	at the beginning of the current line
a	at the current cursor location (text inserted after the currsor)
A	At the end of the current line

The characters don't appear on the screen, but any text typed after them becomes part of the file you are editing.

To leave insert mode and reenter command mode, press **ESC** . For more explanation of modes in **vi** , see "Inserting Text" earlier in this chapter.

CORRECTING TYPING ERRORS

To correct a typing error on the current line, backspace over the mistake then retype the line. Other ways to correct typing mistakes are described in this section.

OPENING A NEW LINE

You can use several methods to open new lines in a file:

PRESS	TO OPEN A NEW LINE...
O	above the current line
o	below the current line

Both commands place you in insert mode; you may enter text immediately.

If you're in insert mode, you can also use the **CR** key to open new lines. To open a line above the current line, move the cursor to the beginning of the line, enter insert mode, then press **CR** . To open a line below the current line, move the cursor to the end of the line, enter insert mode, then press **CR**

REPEATING THE LAST INSERTION

CTRL @ repeats the last insertion. Enter insert mode, then press **CTRL @**

CTRL @ repeats insertions of 128 characters or less. If more than 128 characters were inserted, **CTRL @** does nothing.

For other methods of repeating an insertion, see the next two subsections and the subsection entitled "Moving Text".

INSERTING TEXT FROM OTHER FILES

To insert the contents of another file into the current file, use the read command, ;r.

1. Move the cursor to the line immediately above the intended location of the new material.

2. Type:

   :r *filename*

   where *filename* is the file containing the material to be inserted, and press **CR** .

   The text of *filename* appears on the line below the cursor, and the cursor moves to the first character of the new text. This text is a copy; the original filename still exists.

   To insert selected lines from another file, you first copy them from the original file into a temporary holding place called a buffer, and then insert them into the new file.

1. Use the write command, **:w**, to save the file you're copying to. Do not exit vi.

2. Type:

   :e *filename*

   where *filename* is the file that contains the text you want to copy. Press **CR** .

3. When the file appears on screen, move the cursor to the first line you wish to select.

4. Mark this line by typing:

   mk

5. Move the cursor to the last line of the text you wish to select. Type:

   "ay'k

   This command "yanks" the lines from marked location to the cursor, and places them into buffer "a". They will remain in buffer "a" until you replace them with other lines, or until you exit the editor.

6. Type:

   :e#

   to return to your previous file.

7. Move the cursor to the line above the intended location of the new text, and type:

   "ap

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

You can have 26 buffers named "a", "b"", "c", up to and including "z". To name and select different buffers, replace the "a" in steps 5 and 6 with whatever letter you wish.

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

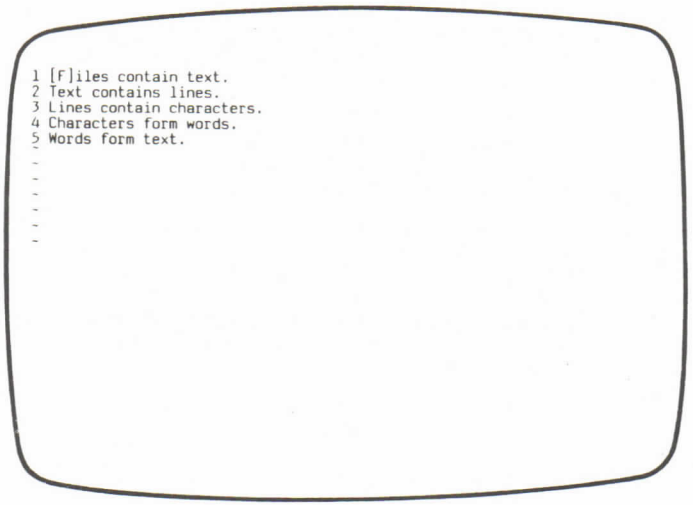
Copying Lines from Elsewhere in the File

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

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

:nu

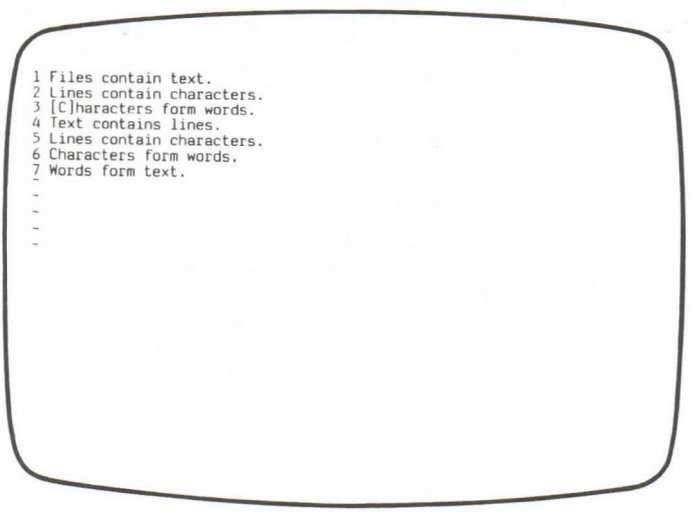
and press **CR** . The line number and the text of that line appear on the status line. To find out the destination line number, move the cursor to the line above where you want the copied text to appear and repeat the **:nu** command. You can also make line numbers appear throughout the file with the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



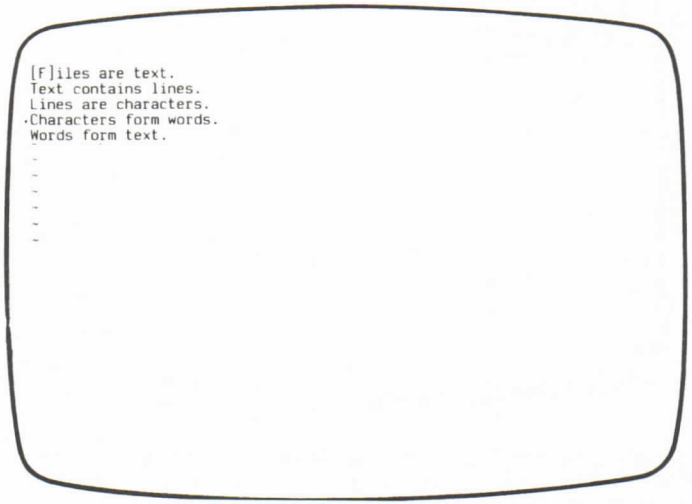
Using the above example, to place copies of lines 3 and 4 between lines 1 and 2, type:

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

The result is:



To insert text several times in different places, you can save it in a buffer, and insert it whenever it is needed. For example, to repeat the first line of the following text after the last line:



1. Move the cursor over the "F" in "Files". Type the following line, which will not appear on your screen:

   "ayy

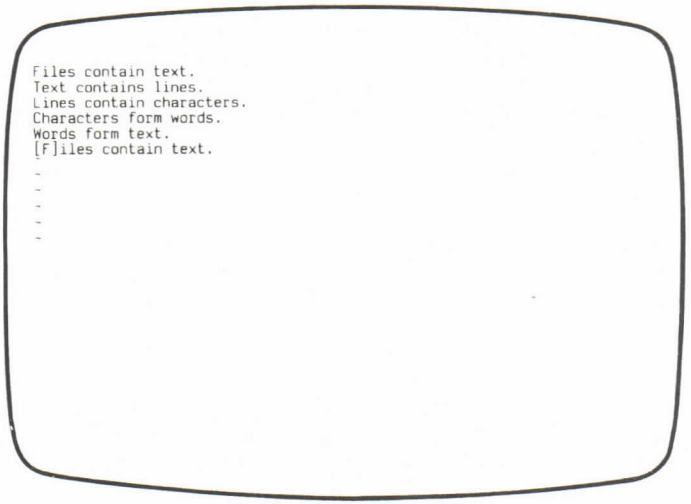
   This "yanks" a copy of the first line into buffer "a". Move the cursor over the "W" in "Words".

2. Type the following line:

   "ap

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

Your screen looks like this:



To yank several consecutive lines, indicate the number of lines you wish to yank after the name of the buffer. For example, to place three lines from the abbve text in the buffer "a", type:

```
"a3yy
```

INSERTING CONTROL CHARACTERS INTO TEXT

Many control characters have special meaning in **vi** , even when typed in insert mode. To remove their special significance, press **CTRL V** before typing the control character. Note that **CTRL J** , **CTRL Q** , and **CTRL S** cannot be inserted as text: **CTRL J** is a newline character; **CTRL Q** and **CTRL S** are meaningful to the operating system, and are trapped by it before they are interpreted by **vi** .

JOINING AND BREAKING LINES

To join two lines, type:

```
J
```

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

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

```
r
```

then press **CR** .

DELETING A CHARACTER: x AND X

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

```
3x
```

To delete three characters preceding the cursor, type:

```
3X
```

DELETING A WORD: dw

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

```
3dw
```

DELETING A LINE: D AND dd

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

```
3dd
```

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

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

```
:200,250d
```

Press **CR** . When the command finishes, the message:

```
50 lines
```

appears on the vi status line, indicating the number of lines deleted.

It is possible to remove lines without displaying line numbers using shorthand "addresses". For example, to remove all lines from the current line to the end of the file, type:

```
:.,$d
```

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

```
:.,+3
```

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

```
:.,-3
```

For more information on using addresses in line-oriented commands, see vi(C) in the XENIX User and System Administrator Reference Manual.

DELETING AN ENTIRE INSERTION

To delete all of the text you just typed, press **CTRL U** while you are in insert mode. The cursor returns to the beginning of the insertion. The text of the original insertion is still displayed, and any text you type replaces it. When you press **ESC** , any text remaining from the original insertion disappears.

DELETING AND REPLACING TEXT

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

Overstriking: r and R

The **r** command replaces the current character with the next character typed. To replace the current character with a "b", for example, type:

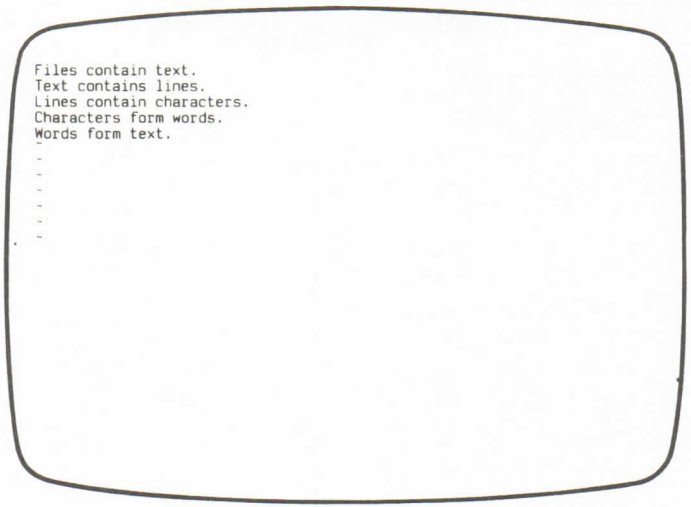
```
rb
```

If a number precedes **r** , that number of characters are replaced with the next character typed. For example, to replace the current character, plus the next three characters, with the letter "b", type:

```
4rb
```

Note that you now have four "b" s in a row.

The **R** lets you overstrike characters up to the end of the current line. To end the replacement, press **ESC** . For example, to replace the second line in the following text with "Spelling is important.":

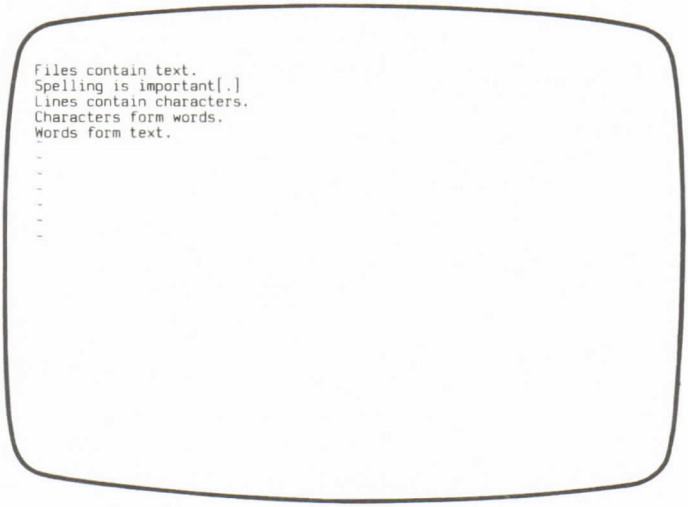


1. Move the cursor over the "T" in "Text".

2. Press **R** , then type:

   Spelling is important.

3. Press **ESC** to end the replacement. If you make a mistake, use the **BKSP** key. Your screen should now look like this:



Substituting: s and S

The s command replaces a specified number of characters, beginning with the current character with text you type in. For example, to substitute "xyz" for the cursor and two characters following it, type:

```
3sxyz
```

The S command deletes a specified number of lines and replaces them with text you type in. You may type in as many new lines of text as you wish; S affects only how many lines are deleted. If no number is provided, one line is deleted. For example, to delete four lines, including the current line, type:

```
4S
```

This differs from the R command. The S command deletes the entire current line; the R command deletes text from the cursor onward.

Replacing a Word: cw

The cw command replaces a word with text you type in. For example, to replace the word with the word "fox", move the cursor over the "b" in "bear". Press:

```
cw
```

A dollar sign appears after the "r" in "bear", marking the end of the text that is being replaced. Type:

fox

and press **CR** . The rest of "bear" disappears and only "fox" remains.

Replacing the Rest of a Line: C

The **C** command replaces text from the cursor to the end of the line. For example, to replace the text of the sentence:

Who's afraid of the big bad wolf?

from "big" to the end, move the cursor over the "b" in "big" and type:

C

A dollar sign ($) replaces the question mark (?) at the end of the line. Type the following:

little lamb?

Press **ESC** . The remaining text from the original sentence disappears.

Replacing a Whole Line: cc

The **cc** command deletes a specified number of lines, regardless of the location of the cursor, and replaces them with text you type in. If no number is given, the current line is deleted.

Replacing a Particular Word on a Line

If a word occurs several times on one line, it is often convenient to use a line-oriented command to replace it. For example, to replace the word "removing" with "deleting" in the following sentence:

In vi, removing a line is as easy as removing a letter.

Make sure the cursor is at the beginning of that line, and type:

:s/removing/deleting/g

Press **CR** . This line-oriented command means "Substitute (s) for the word removing the word deleting, everywhere it occurs on the current line (g)". If you don't include a **g** at the end, only the first occurrence of removing is changed.

For more information on using line-oriented commands to replace text, see the subsection entitled "Pattern Matching".

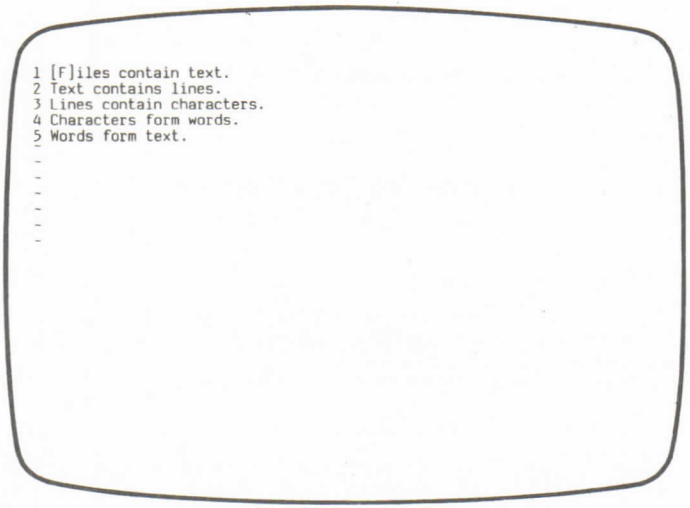
MOVING TEXT

To move a block of text from one place in a file to another, you can use the line-oriented **m** command. You must know the line numbers of your file to use this command. The linenumber option displays line numbers. To set this option, enter command mode, then type:

```
set linenumber
```

Line numbers will appear to the left of your text.

The following example uses the linenumber option:



To insert lines 2 and 3 between lines 4 and 5, type:

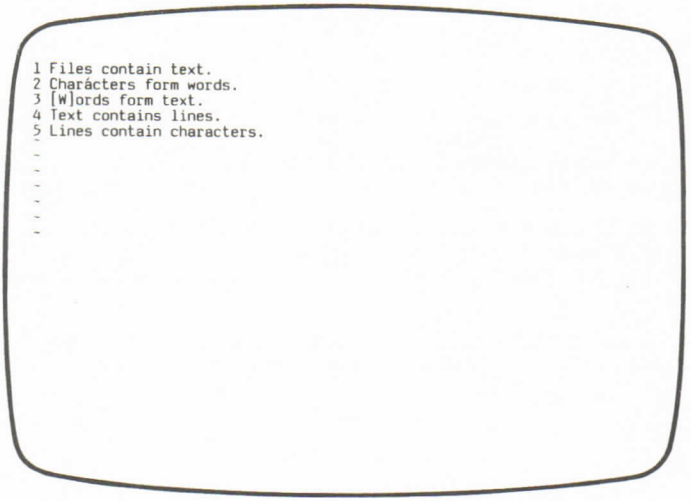
```
:2,3m4
```

Your screen should look like this:

To place line 5 after line 2, type:

```
:5m2
```

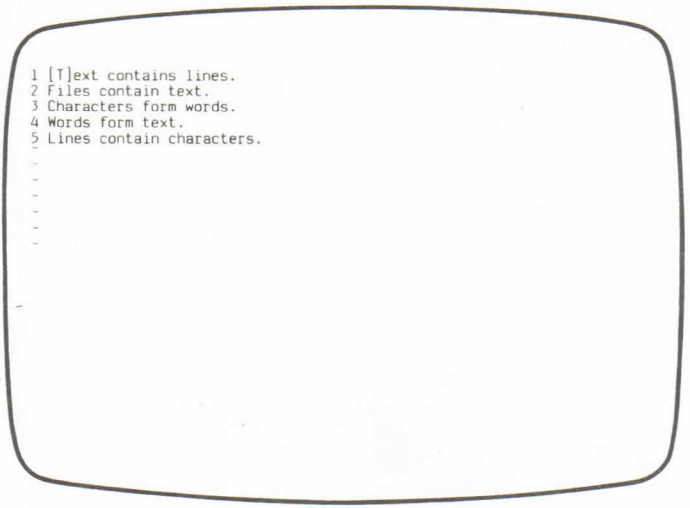
After moving, your screen should look like this:



To make line 4 the first line in the file, type:

```
:4m0
```

Your screen should look like this:

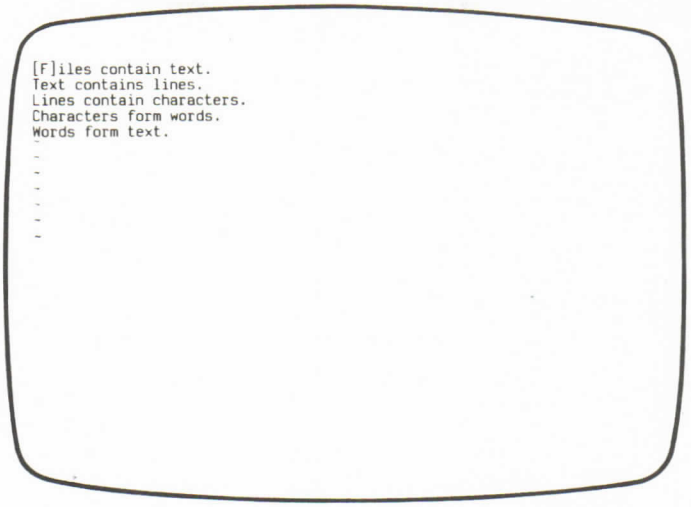


You can also delete text into a temporary storage place, called a buffer, and insert it wherever you wish. When text is deleted it is placed in a "delete buffer". There are nine delete buffers.

The first buffer always contains the most recent deletion. In other words, the first deletion in a given editing session goes into buffer 1. The second deletion also goes into buffer 1, and pushes the contents of the old buffer 1 into buffer 2. The third deletion goes into buffer 1, pushing the contents of buffer 2 into buffer 3, and the contents of buffer 1 into buffer 2. When buffer 9 has been used, the next deletion pushes the current text of buffer 9 off the stack and it disappears.

Text remains in the delete buffers until it is pushed off the stack, or until you quit the editor, so it is possible to delete text from one file, change files without leaving the editor, and place the deleted text in another file.

Delete buffers are particularly useful when you wish to remove text, store it, and put it somewhere else. Using the following text as an example:



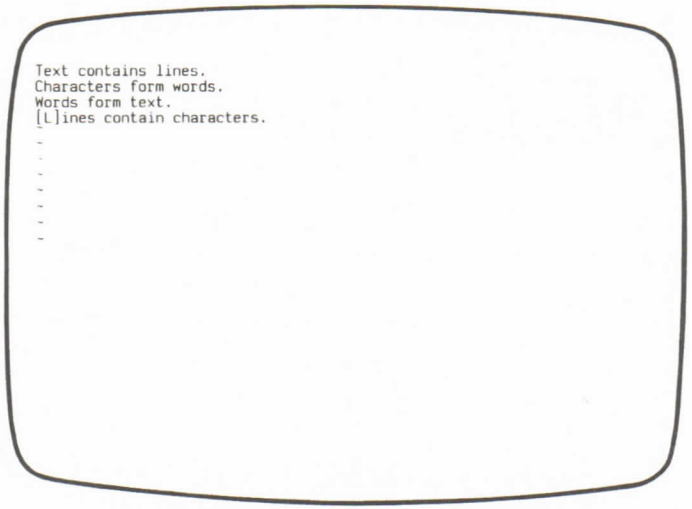
Delete the first line by typing:

```
dd
```

Delete the third line the same way. Now move the cursor to the last line in the example and type:

```
"1p
```

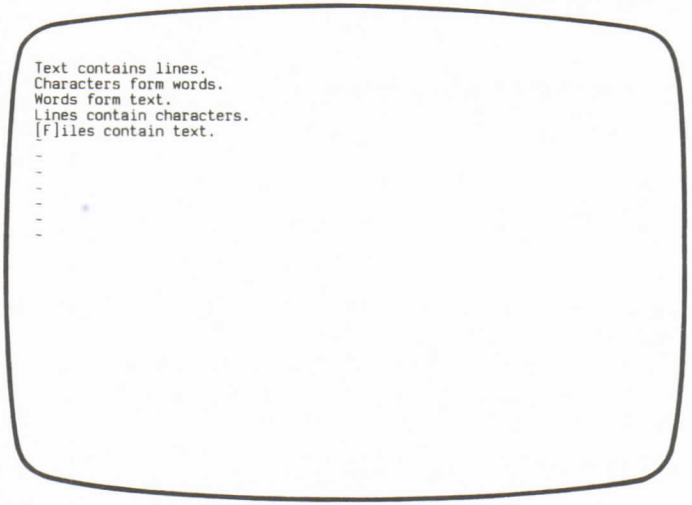
The line from the second deletion appears:



Now type:

"2p

The line from the first deletion appears:



Since the text remains in a buffer until it is either pushed off the

stack or until you quit the editor, you may use it as many times as you wish.

It is also possible to place text in named buffers. For information on how to create named buffers, see the subsection entitled "Inserting Text from Other Files".

SEARCHING: / AND ?

vi lets you search forward and backward in a file for patterns. To search forward:

1. Press the slash (/) key. The slash appears on the status line.

2. Type the characters you wish to search for.

3. Press **CR** .

If the specified pattern exists, the cursor will move to the first character of the pattern. For example, to search forward in the file for the word "account", type:

```
/account
```

Press **CR** . The cursor appears on the first character of the pattern.

To search backward through a file, use ? instead of / to start the search. For example, to find all occurrences of "account" above the cursor, type:

```
?account
```

To search for a pattern containing any of the special characters (. * \ [ ] ~ $ and ^), you must precede each special character on the status line with backslash. For example, to find the pattern "U.S.A.", type:

```
/U \. S \. A \./
```

To continue searching for a pattern, type:

```
n
```

after each search. You can continue to use **n** until you specify a new pattern or quit the editor.

Because XENIX is case sensitive, vi searches for exactly what you type. To have XENIX disregard case in a search command, you can set the ignorecase option before initiating the search:

```
:set ignorecase
```

By default, searches "wrap around" the file. That is, if a search starts in the middle of a file, when vi reaches the end of the file it will "wrap around" to the beginning, and continue until it returns to where

the search began.

If you do not want searches to wrap around the file, you can change the wrapscan option setting. Type:

```
:set nowrapscan
```

and press **CR** to prevent searches from wrapping. For more information about setting options, see the section entitled "Setting up your Environment".

SEARCHING AND REPLACING

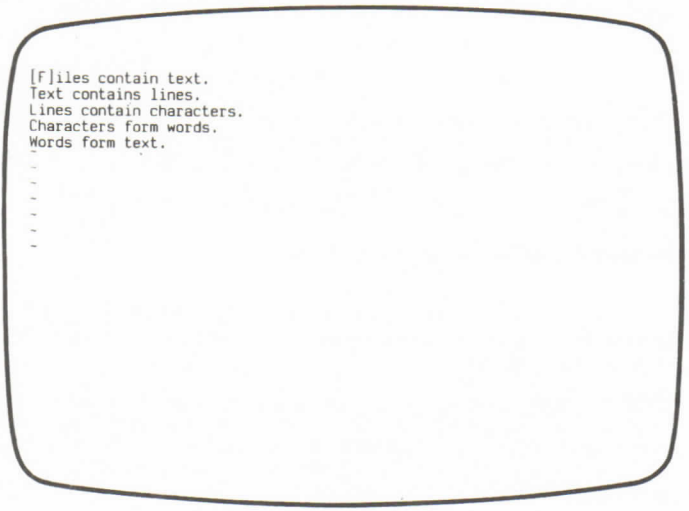
The search and replace commands allow you to perform complex changes to a file in a single command.

The syntax of a search and replace command is:

g/ *pattern1* /**s**/ [*pattern2*] / [*options*]

Brackets indicate optional parts of the command line. The **g** tells the computer to execute the replacement on every line in the file. Otherwise the replacement would occur only on the current line. The options are explained in the following sections.

To explain these commands we will use the example file from the demonstration:



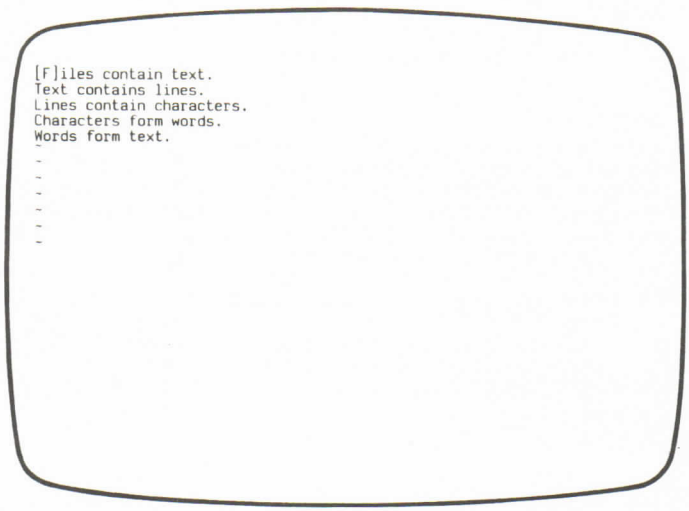
Replacing a Word

To replace the word "contain" with the word "are" throughout the file, type the following command:

```
:g/contain /s//are /g
```

This command says "On each line of the file (g), find "contain" and substitute for that word (s//) the word "are", everywhere it occurs on that line (the second g)". Note that a space is included in the search pattern for "contain"; without the space "contains" would also be replaced.

After the command executes, your screen should look like this:



Printing All Replacements

To replace "contain" with "are" throughout the file, then print every line changed, use the **p** option:

```
:g/contain /s//are /gp
```

Press **CR** . After the command executes, each line in which "contain" was replaced by "are" is printed on the lower part of the screen. To remove these lines, redraw the screen by pressing **CTRL L** .

Confirming Replacements

Sometimes you may not want to replace every instance of a given pattern. The **c** option displays every occurrence of pattern and waits for you to confirm that you want to make the substitution. If you press "y" the substitution takes place; if you press **CR** the next instance of "pattern" is displayed.

To run this command on the example file, type:

```
:g/contain/s//are/gc
```

Press **CR** . The first instance of "contain" appears on the status line:

```
Files con^^^^^^^ text.
```

Press **y** . then **CR** . The next occurrence of "contain" appears.

PATTERN MATCHING

Search commmands often require, in addition to the characters you want to find, a context in which you want to find them. For example, you may want to locate every occurrence of a word at the beginning of a line. vi provides several special characters that specify particular contexts.

Matching the Beginning of a Line

When a caret(^) is placed at the beginning of a pattern, only patterns found at the beginning of a line are matched. For example, the following search pattern only finds "text" when it occurs as the first word on a line:

```
/^text/
```

To search for a caret that appears as text you must precede it with a backslash (\).

Matching the End of a Line

When a dollar sign ($) is placed at the end of a pattern, only patterns found at the end of a line are matched. For example, the following search pattern only finds "text" when it occurs as the last word on a line:

```
/text$/
```

To search for a dollar sign that appears as text you must precede it with a backslash (\).

Matching Any Single Character

When used in a search pattern, the period (.) matches any single character except the newline character. For example, to find all words that end with "ed", use the following pattern:

```
/.ed /
```

Note the space between the "d" and the backslash.

To search for a period in the text, you must precede it with a backslash (\).

Matching a Range of Characters

A set of characters enclosed in square brackets matches any single character in the range designated. For example, the search pattern:

```
/[a-z]/
```

finds any lowercase letter. The search pattern:

```
/[aA]pple/
```

finds all occurrences of "apple" and "Apple".

To search for a bracket that appears as text, you must precede it with a backslash ().

Matching Exceptions

A caret (^) at the beginning of string matches every character except those specified in string. For example, the search pattern

```
[^a-z]
```

finds anything but a lowercase letter or a newline.

Matching the Special Characters

To place a caret, hyphen or square bracket in a search pattern, precede it with a backslash. To search for a caret, for example, type:

```
/\^/
```

If you need to search for many patterns that contain special characters, you can reset the **magic** option. To do this, type:

```
:nomagic
```

This removes the special meaning from the characters ., \, $, [and]. You can include them in search and replace commands without a preceding

backslash. Note that the special meaning cannot be removed from the special characters asterisk (*) and caret (^); these must always be preceded by a backslash in searches.

To restore **magic** , type:

```
:set  magic
```

UNDOING A COMMAND: u

Any editing command can be reversed with the **undo** (u) command. **u** works on both screen-oriented and line-oriented commands. For example, if you have deleted a line and then decide you wish to keep it, press **u** and the line will reappear.

u undoes only the last command. For example, if you make a global search and replace, then delete a few characters with the **x** command, pressing **u** will undo the deletions but not the global search and replace.

REPEATING A COMMAND: .

To repeat a screen-oriented **vi** command, use the repeat (.) command. For example, if you have deleted two words by typing:

```
2dw
```

you may repeat this command as many times as you wish by pressing the period key (.). Cursor movement does not affect the repeat command, so you may repeat a command as many times and in as many places in a file as you wish.

The repeat command repeats only the last **vi** command.

LEAVING THE EDITOR

There are several ways to exit the editor and save any changes you may have made to the file. One way is to type:

```
:x
```

and press **CR** . This command replaces the old copy of the file with the new one you have just edited, quits the editor, and returns you to the XENIX shell. Similarly, if you type:

```
ZZ
```

the same thing happens, except the old copy file is written out only if you have made any changes. Note that the ZZ command is not preceded by a colon, and is not echoed on the screen.

To leave the editor without saving any changes you have made to the file,

type:

```
:q!
```

The exclamation point tells **vi** to quit unconditionally. If you leave out the exclamation point:

```
:q
```

vi will not let you quit. You will see the error message:

```
No write since last change (:quit! overrides)
```

This message tells you to use **:q!** if you really want to leave the editor without saving your file.

Saving a File Without Leaving the Editor

There are many occasions when you must save a file without leaving the editor, such as when starting a new shell, or moving to another file. Before you can perform these tasks you must first save the current file with the **write** command:

```
:w
```

You do not need to type the name of the file; **vi** remembers the name you used when you invoked the editor. If you invoked **vi** without a filename, you may name the file by typing:

```
:w filename
```

where *filename* is the name of the new file.

EDITING A SERIES OF FILES

Entering and leaving **vi** for each new file takes time, particularly on a heavily used system, or when you are editing large files. If you have many files to edit in one session, you can invoke **vi** with more than one filename, and thus edit more than one file without leaving the editor, as in:

```
vi file1 file2 file3 file4 file5 file6
```

You can also invoke **vi** using the special characters as abbreviations. For example, to invoke **vi** on the above files without typing each name, type:

```
vi file*
```

This invokes **vi** on all files that begin with the letters "file". You can plan your filenames to save time in later editing. For example, if you are writing a document that consists of many files, it would be wise to give each file the same filename extension, such as ".s". Then you can

invoke **vi** on the entire document:

```
vi *.s
```

You can also invoke **vi** on a selected range of files:

```
vi [3-5]*.s
```

or

```
vi [a-h]*
```

To invoke **vi** on all files that are five letters long, and have any extension:

```
vi ?????.*
```

For more information on using special characters, see Chapter 3.

When you invoke **vi** with more than one filename, you will see the following message when the first file appears on the screen:

x **files to edit**

After you have finished editing a file, save it with the write command (**:w**), then go to the next file with the next command:

```
:n
```

The next file appears, ready to edit. It is not necessary to specify a filename; the files are invoked in alphabetical (or numerical, if the filenames begin with numbers) order.

If you forget what files you are editing, type:

```
:args
```

The list of files appears on the status line. The current file is enclosed in square brackets.

To edit a file out of order, such as file4 after file2, type:

```
:e file4
```

instead of using the next command. If you type:

```
:n
```

after you finish editing file4, you will go back to file3.

If you wish to start again from the beginning of the list, type:

```
:rew
```

To discard the changes you made and start again at the beginning, type:

```
:rew!
```

EDITING A NEW FILE WITHOUT LEAVING THE EDITOR

You can start editing another file anywhere on the XENIX system without leaving **vi**. This saves time when you wish to edit several files in one session that are in different directories, or even in the same directory. For example, if you have finished editing /usr/joe/memo and you wish to edit /usr/mary/letter, first save the file memo with the write (**:w**) command, then type:

```
:e  /usr/mary/letter
```

/usr/mary/letter appears on your screen just as though you had left **vi**.

Note You must write out your file with the write (**:w**) command if you want to save the changes you have made. If you try to edit a second file without writing out the first file, the message:

```
No write since last change (:e! overrides)
```

appears. If you use **:e!** all your changes to the first file are discarded.

If you want to switch back and forth between two files, **vi** remembers the name of the last file edited. Using the previous example, if you wish to go back and edit the file /usr/joe/memo after you have finished with /usr/mary/letter, type:

```
:e#
```

The cursor is positioned in the same location it was when you first saved /usr/joe/memo.

LEAVING THE EDITOR TEMPORARILY: Shell Escapes

You can execute any XENIX command from within **vi** using the shell "escape from vi" command, **!**. For example, to find out the date and time, type:

```
:!date
```

The exclamation point sends the remainder of the line to the shell to be executed, and the date and time appear on the **vi** status line. You can use the **!** to perform any XENIX command. To send mail to joe without leaving the editor, type:

```
:!mail joe
```

Type your message and send it. (For more information about the XENIX mail system, see Chapter 5, "Using mail".) After you send it, the

message:

```
[Hit return to continue]
```

appears. Press **CR** to continue editing.

If you want to perform several XENIX commands before returning to the editor, you can invoke a new shell:

```
:!sh
```

The XENIX prompt appears. You may execute as many commands as you like. Press **CTRL D** to terminate the new shell and return to your file.

If you have not written out your file before a shell escape, you will see the message:

```
[No write since last change]
```

It is a good idea to save your file with the write (**:w**) command before executing an escape, just in case something goes wrong. However, once you become an experienced **vi** user, you may wish to turn off this message. To turn off the **No write** message, reset the warn option:

```
:set  nowarn
```

For more information about setting options in **vi** , see the section entitled "Setting up Your Environment".

PERFORMING A SERIES OF LINE-ORIENTED COMMANDS: Q

If you have several line-oriented commands to perform, you can place yourself temporarily in line-oriented mode by typing:

```
Q
```

while you are in command mode. A colon prompt appears on the status line.

Commands executed in this mode cannot be undone with the **u** command, nor do they appear on the screen until you re-enter normal **vi** mode. To re-enter normal **vi** mode, type:

```
vi
```

FINDING OUT WHAT FILE YOU ARE IN

If you forget what file you are editing, press **CTRL G** while you are in command mode. A line similar to the following appears appears on the status line:

```
"memo" [Modified] line 12 of 100 --12%--
```

From left to right, the following information is displayed:

- The name of the file
- Whether or not the file has been modified
- The line number the cursor is on
- The number of lines in the file
- Your location in the file (expressed as a percentage)

This command is also useful when you need to know the line number of the current line for a line-oriented command.

The same information can be obtained by typing:

```
:file
```

or:

```
:f
```

FINDING OUT WHAT LINE YOU ARE ON

To find out what line of the file you are on, type:

```
:nu
```

and press **CR** . This command displays the current line number and the text of the line.

SOLVING COMMON PROBLEMS

The following is a list of common problems that you may encounter when using **vi**, along with the probable solution.

- I don't know which mode I'm in.

  Press **ESC** until the bell rings. When the bell rings you are in command mode.

- I can't get out of a subshell.

Press **CTRL D** to exit any subshell. If you have created more than one subshell (not a good idea, usually), keep pressing **CTRL D** until you see the message:

[Hit return to continue]

- I made an inadvertent deletion (or insertion).

  Press **u** to undo the last delete or insert command.

- There are extra characters on my screen.

  Press **CTRL L** to redraw the screen.

- When I type, nothing happens.

  vi has crashed and you are now in the shell with your terminal characteristics set incorrectly. To reset the keyboard, slowly type:

  stty sane

  then press **CTRL J** or **LINEFEED** . Pressing **CTRL J** instead of **CR** is important here, since it is quite possible that the **CR** key will not work as a newline character. To make sure that other terminal characteristics have not been altered, log off, turn your terminal off, turn your terminal back on, and then log back in. This should guarantee that your terminal's characteristics are back to normal. This procedure may vary somewhat depending on the terminal.

- The system crashed while I was editing.

  Normally, **vi** will inform you (by sending you mail) that your file has been saved before a crash. The file can be recovered by typing:

  vi -r *filename*

  If **vi** was unable to save the file before the crash, it is irretrievably lost.

- I keep getting a colon on the status line when I press **CR** .

  You are in line-oriented command mode. Type:

  vi

  to return to normal **vi** command mode.

- I get the error message **Unknown terminal type [Using open mode]** when I invoke **vi**.

  Your terminal type is not set correctly. To leave open mode, press **ESC** , then type:

  :wq

and press **CR** . Turn to the subsection entitled "Setting the Terminal Type" for information on how to set your terminal type correctly.

SETTING UP YOUR ENVIRONMENT

You can set a number of options to affect your terminal type, how files and error messages are displayed on your screen, and how searches are performed. These options can be set with the **set** command while you are editing, or they can be placed in the **vi** startup file, exrc. The following sections describe the most commonly used options and how to set them. There is a complete list of options in **vi(C)** in the XENIX User and System Administrator Reference Manual.

SETTING THE TERMINAL TYPE

Before you can use **vi** , you must set the terminal type (if this has not already been done for you) by defining the **TERM** variable in your .profile or .login file. The **TERM** variable is a number that tells the operating system what type of terminal you are using. To determine this number, you must find out what type of terminal you are using. Then look up this type in **terminals (M)** in the XENIX Reference Manual. If you cannot find your terminal type or its number, consult your system administrator.

For these examples, we will suppose that you are using an HP 2621 terminal. For the HP 2621, the **TERM** variable is "2621". How you define this variable depends on which shell you are using. You can usually determine which shell you are using by examining the prompt character. The Bourne shell prompts with a dollar sign ($); the C shell prompts with a percent sign (%).

Setting the TERM variable: The Visual Shell

If you are using the Visual Shell, the terminal type has already been set, and you do not need to change it.

Setting the TERM variable:The Bourne Shell

To set your terminal type to "2621", place the following commands in the file .profile:

```
TERM=2621
export TERM
```

Setting the TERM variable: The C Shell

To set your terminal type to "2621" for the C shell, place the following command in the file .login:

```
setenv TERM 2621
```

SETTING OPTIONS: set

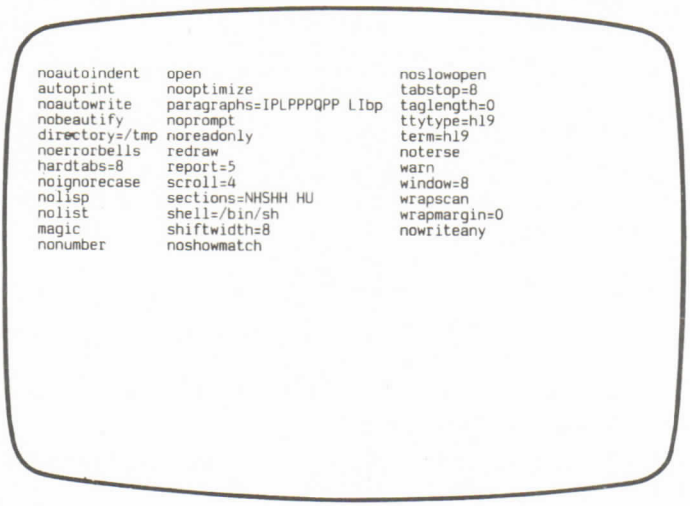
The **set** command displays option settings and lets you set options.

Listing the Available Options

To get a list of the options available to you and how they are set, type:

```
:set all
```

Your display should look similar to this:



This chapter discusses only the most commonly used options. For information about the options not covered in this chapter, see **vi(C)** in the XENIX User and System Administrator Reference Manual.

Setting an Option

To set an option, use the **set** command. For example, to set the ignorecase option so that case is not ignored in searches, type:

```
:set noignorecase
```

DISPLAYING TABS AND END-OF-LINE: list

List causes the "hidden" characters and end-of-line to be displayed. The default setting is nolist. To display these characters, type:

```
:set list
```

Your screen is redrawn. The dollar sign ($) represents end-of-line and **CTRL I** (^I) represents the TAB character.

IGNORING CASE IN SEARCH COMMANDS: ignorecase

By default, case is significant in search commands. To disregard case in searches, type:

```
:set ignorecase
```

To change this option, type:

```
:set noignorecase
```

DISPLAYING LINE NUMBERS: number

It is often useful to know the line numbers of a file. To display these numbers, type:

```
:set number
```

This redraws your screen. Numbers appear to the left of the text. To remove line numbers, type:

```
:set nonumber
```

PRINTING THE NUMBER OF LINES CHANGED: report

The **report** option tells you the number of lines modified by a line-oriented command. For example:

```
:set report=1
```

reports the number of lines modified, if more than one line is changed. The default setting is:

```
report=5
```

which reports the number of lines changed when more than five lines are modified.

CHANGING THE TERMINAL TYPE: term

If you are logged in on a terminal that is a different type than the one you normally use, you can check the terminal type setting by typing:

```
:set term
```

SHORTENING ERROR MESSAGES: terse

After you become experienced with **vi** , you may want to shorten your error messages. To change from the default (**noterse**), type:

```
:set terse
```

As an example of the effect of **terse** , when **terse** is set the message:

```
No write since last change, quit! overrides
```

becomes:

```
No write
```

TURNING Off WARNINGS: warn

After you become experienced with **vi** , you may want to turn off the error message that appears if you have not written out your file before a shell escape (**:!**) command. To turn these messages off, type:

```
:set nowarn
```

PERMITTING SPECIAL CHARACTERS IN SEARCHES: nomagic

The **nomagic.** option allows the inclusion of the special characters (. \ $ [ ]) in search patterns without a preceding backslash. This option does not affect caret (^) or asterisk (*); they must be preceded by a backslash in searches regardless of **magic.** To set **nomagic** , type:

```
:set nomagic
```

LIMITING SEARCHES: wrapscan

By default, searches in vi "wrap" around the file until they return to the place they started. To save time you may want to disable this feature. Use the following command:

```
:set nowrapscan
```

When this option is set, forward searches go only to the end of the file, and backward searches stop at the beginning.

TURNING ON MESSAGES: mesg

When you invoke vi , write permission to your screen is automatically turned off, preventing write messages from appearing. If you wish to receive write messages while in vi , reset this option as follows:

```
:set mesg
```

To remove the message from your display you must press CTRL L .

CUSTOMIZING YOUR ENVIRONMENT: The .exrc File

Each time vi is invoked, it reads commands from the file named .exrc in your home directory. This file sets your preferred options so that they do not need to be set each time you invoke vi. A sample .exrc file follows:

```
set number
set ignorecase
set nowarn
set report=1
```

Each time you invoke vi with the above options, your file is displayed with line numbers, case is ignored in searches, warnings before shell escape commands are turned off, and any command that modifies more than one line will display a message indicating how many lines were changed.

COMMAND SUMMARY

The following tables contain all the basic the commands discussed in this chapter.

ENTERING VI

TYPING THIS:	DOES THIS:
vi *file*	starts at line 1
vi + *n file*	starts at line *n*
vi + *file*	starts last line
vi +/ *pattern file*	starts at *pattern*
vi -r *file*	recovers *file* after a system crash

CURSOR MOVEMENT

PRESSING THIS:	DOES THIS:
h	moves one space left
l	moves one space right
SPACEBAR	moves one space right
w	moves one word right
b	moves one word left
k	Moves one line up
j	moves one line down
CR	moves one line down
)	moves to end of sentence
(	moves to beginning of sentence
}	moves to beginning of paragraph
{	moves to end of paragraph
CTRL W	moves to first character of insertion
CTRL U	scrolls up 1/2 screen
CTRL D	scrolls down 1/2 screen
CTRL F	scrolls down one screen
CTRL B	scrolls up one screen

INSERTING TEXT

PRESSING	STARTS INSERTION:
i	before the cursor
I	before first character on the line
a	after the cursor
A	after last character on the line
o	on the following line
O	on the previous line
r	on the current character; replaces one character only
R	on the current character; replaces until ESC

DELETE COMMANDS

COMMAND	FUNCTION
dw	deletes a word
d0	deletes to beginning of line
d$	deletes to end of line
n dw	deletes *n* words
dd	deletes the current line
n dd	deletes *n* lines

CHANGE COMMANDS

COMMAND	FUNCTION
cw	changes one word
n cw	changes *n* words
cc	changes current line
n cc	changes *n* lines

SEARCH COMMANDS

Command	Function	Example
/and	Finds the next occurance of "and	and, stand, grand
?and	Finds the previous occurence of "and"	and, stand, grand
/^The	Finds next line that starts with "The"	The, Then, There
/[bB]ox/	Finds the next occurence of "box" or "Box"	
n	Repeats the most recent search, in the same direction	

SEARCH AND REPLACE COMMANDS

Command	Result	Example
:s/pear/peach/g	All pears become peach on the current line	
:1,$s/file/directory	Replaces file with directory from line 1 to the end.	filename becomes directoryname
:g/one/s//1/g	Replaces every occurrence of one with 1.	one becomes 1, oneself becomes 1self, someone becomes some1

PATTERN MATCHING: SPECIAL CHARACTERS

THIS CHARACTER:	MATCHES:
^	beginning of a line
$	end of a line
.	any single character
[]	a range of characters

LEAVING VI

COMMAND	RESULT
:w	writes out the file
:x	writes out the file, quits vi
:q!	quits vi without saving changes
:!*command*	executes *command*
:!sh	forks a new shell
!!*command*	executes *command* and places output on current line
:e *file*	edits *file* (save current file with :w first)

OPTIONS

THIS OPTION:	DOES THIS:
all	lists all options
term	sets terminal type
ignorecase	ignores case in searches
list	displays tab and end-of-line characters
number	displays line numbers
report	prints number of lines changed by a line-oriented command
terse	shortens error messages
warn	turns off "no write" warning before escape
nomagic	allows inclusion of special characters in search patterns without a preceding backslash

THIS OPTION:	DOES THIS:
nowrapscan	prevents searches from wrapping around the end or beginning of a file
mesg	permits display of messages sent to your terminal with the **write** command

11. USING ed

ABOUT THIS CHAPTER

This chapter serves as an introduction to the XENIX V line editor, **ed** .

CONTENTS

INTRODUCTION

ed is a text editor used to create and modify text. The text is normally a document, a program, or data for a program. Note that the line editor ex is very similar to ed; you can use this chapter as an introduction to ex as well as to ed.

DEMONSTRATION

This section leads you through a simple session with ed, giving you a feel for how it is used and how it works. To begin the demonstration, invoke ed by typing:

```
ed
```

This invokes the editor and begins your editing session. An asterisk (*) prompts for commands to be entered. Initially, you are editing a temporary file called the editing buffer.

Typically, the first thing you will want to do with an empty buffer is add text to it. For example, after the prompt, type:

```
a
this is line 1
this is line 2
this is line 3
this is line 4
  CTRL D
```

This appends four lines of text to the buffer. To view these lines on your screen, type:

```
1,4p
```

where the "1,4" specifies a line number range and the p command "prints" the specified lines on the screen.

Now type:

```
2p
```

to view line number two. Then type:

```
p
```

This prints out the current line on the screen, which happens to be line number two. By default, most ed commands operate on only the current line.

BASIC CONCEPTS

This section illustrates some of **ed** ' s basic concepts.

THE EDITING BUFFER

Each time you invoke **ed**, XENIX allocates an area of computer memory in which you will perform all of your editing operations. This area is called the editing buffer. XENIX places a copy of the file you specify into the buffer for editing. Only when you write out your file do you affect the original file.

COMMANDS

You enter commands by typing them at your keyboard and pressing **CR** (Although our command examples do not include **CR** , its presence is assumed.) Most commands are single characters that can be preceded by the specification of a line number or a line number range. By default, most commands operate on the current line. Many commands take filename or string arguments that are used by the command when it is executed.

If you make an error while entering a command, **ed** will display the message:

```
?
error message
```

LINE NUMBERS

Whenever you execute a command that changes the number of lines in the editing buffer, **ed** immediately renumbers the lines. Many editing commands will take either single line numbers or line number ranges as prefixing arguments. These arguments specify the actual lines in the editing buffer that are to be affected by the given command. By default, a special line number called **dot** specifies the current line.

TASKS

This section discusses the tasks you perform in everyday editing.

ENTERING THE EDITOR

To enter **ed**, type:

```
ed filename
```

where *filename* is the name of a new or existing file.

APPENDING TEXT: a

The **append** command lets you add text to the buffer. To use append, type:

```
a
```

followed by the desired text. To stop appending, type a period on a line by itself. For example:

```
a
Now is the time
for all good men
to come to the aid of their party.
.
```

The period (.) tells ed that you have finished appending. (You can also use **CTRL D** , but we will use the period throughout this discussion.) If **ed** doesn't respond, type an extra line with just a period (.) on it.

After appending is complete, the buffer contains the following three lines:

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

The "a" and period don't appear, because they are not text.

WRITING OUT A FILE: w

The **write** command lets you write out the contents of the editing buffer into a file, overwriting the file's previous contents. For example, to save the text in a file named "text" , type:

```
w text
```

ed responds by printing the number of characters it has written out. (Spaces and the newline character at the end of each line are included in the character count.) Writing out a file just makes a copy of the text -- the buffer's contents are not disturbed, so you can go on adding text to

it. If you invoked **ed** with the command "ed *filename* , then by default a **w** command by itself will write the buffer out to *filename* .

No change in the contents of a file takes place until you give a **w** command. It's a good idea to write out the text to a file from time to time as you create it. If the system crashes, you will lose all the text in the buffer, but still have access to any text that was written out to a file.

LEAVING THE EDITOR: q

To terminate an **ed** session, save the text you're working on by writing it to a file using the **w** command, then type:

```
q
```

The system responds with the XENIX prompt character. If you try to quit without writing out the file **ed** will print:

```
?
```

At that point, write out the text if you want to save it; if not, typing another "q" will get you out of the editor.

EDITING A NEW FILE: e

To read existing text into your editing buffer, use the **edit** command, which places the entire contents of a file in the buffer. For example, the **edit** command lets you edit text you have previously saved with the **w** command.

If you had saved the three lines "Now is the time", etc., with a **w** command in an earlier session, the command:

```
e text
```

would place the entire contents of the file *text* into the buffer and respond with:

```
68
```

which is the number of characters in text. If anything is already in the buffer, it is deleted first.

If you use the **e** command to read a file into the buffer, you don't need to use a filename after a subsequent **w** command. **ed** remembers the last filename used in an **e** command, and **w** will write to this file. Thus, a good way to operate is this:

```
ed
e file
[editing session]
w
q
```

This way, you can type **w** from time to time and be secure in the knowledge that if you typed the filename right in the beginning, you are writing out to the proper file each time.

CHANGING THE FILE NAME WRITTEN OUT: f

To find out the last file written to, use the **file(f)** command. Just type **f** without a filename. You can also change the name of the remembered filename with **f**. Thus a useful sequence is:

```
ed precious
f junk
```

which gets a copy of the file named precious , then uses **f** to save the text in the file junk. The original file will be preserved as precious.

READING IN A FILE: r

To read a file into the buffer without destroying what is already there, use the **read(r)** command. The command:

```
r text
```

reads the file text into your editing buffer and adds it to the end of whatever is already in the buffer. This function is useful for combining files.

Like the **w** and **e** commands, after the reading operation is complete, **r** prints the number of characters read in.

DISPLAYING LINES ON THE SCREEN: p

Use the **print(p)** command to print the contents of the editing buffer (or parts of it) on the terminal screen. Specify the lines where you want printing to begin and where you want it to end, separated by a comma and followed by the letter "p". Thus, to print the first two lines of the buffer (that is, lines 1 through 2), type:

```
1,2p
```

responds with:

```
Now is the time
for all good men
```

To print all the lines in the buffer, use the shorthand symbol of the

dollar sign ($) to indicate the last line in the buffer:

```
1,$p
```

This will print all the lines in the buffer (from line 1 to the last line). To stop printing before it is finished, press the **INTERRUPT** key. **ed** then displays:

```
?
interrupt
```

and waits for the next command.

To print the last line of the buffer, type:

```
$p
```

To print a single line, type just the line number. If you type:

```
$
```

ed prints the last line of the buffer.

You can also use **$** in combinations like:

```
$-1,$p
```

which prints the last two lines of the buffer. This helps when you want to see how far you are in your typing.

The next step is to use address arithmetic to combine the line numbers like dot (.) and dollar sign ($) with plus (+) and minus (-). (Note that "dot" is shorthand for the current line, and is discussed in the next section.) Thus:

```
$-1
```

prints the next to last line of the current file (that is, one line before the line $). For example, to recall how far you were in a previous editing session:

```
$-5,$p
```

prints the last six lines. (Be sure you understand why it's six, not five.) If there aren't six lines in the file, you'll get an error message.

The command:

```
.-3,.+3p
```

prints from three lines before the current line (line dot) to three lines after. The plus (+) can be omitted:

```
.-3,.3p
```

is identical in meaning.

Another area in which you can save typing effort in specifying lines is to use plus and minus as line numbers by themselves. For example:

```
-
```

by itself is a command to move back one line in the file. In fact, you can string several minus signs together to move back that many lines. For example:

```
---
```

moves back three lines, as does:

```
-3
```

Thus:

```
-3,+3p
```

is also identical to:

```
?.-3p+3p
```

DISPLAYING THE CURRENT LINE: dot (.)

Suppose your editing buffer contains the following six lines:

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

If you type:

```
1,3p
```

ed displays:

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

Try typing:

```
p
```

This prints:

```
to come to the aid of their party.
```

which is the third line of the buffer. In fact, it is the last (most recent) line that you have done anything with. You can repeat this p

command without line numbers, and **ed** will continue to print line 3.

This happens because **ed** maintains a record of the last line that you did anything to (in this case, line 3, which you just printed). The line most recently acted on is referred to with a period (.) and is called **dot**. **dot** is a line number in the same way that dollar ($) is; it means "the current line", or loosely, "the line you most recently did something to". You can use it in several ways. One possibility is to type:

```
.,$p
```

This will print all the lines from (and including) the current line clear to the end of the buffer. In our example these are lines 3 through 6.

Some commands change the value of **dot** , while others do not. The **p** command sets **dot** to the number of the last line printed. In the example above, **p** sets **dot** to 6.

dot is often used in combinations like this one:

```
.+1
```

Or equivalently:

```
.+1p
```

This means "print the next line" and is one way of stepping slowly through the editing buffer. You can also type:

```
.-1
```

This means "print the line before the current line". This enables you to go backwards through the file if you wish. Another useful command is something like:

```
.-3,.-1p
```

which prints the previous three lines.

Don't forget that all of these change the value of **dot**. To print the value of **dot**, type:

```
.=
```

Essentially, **p** can be preceded by zero, one, or two line numbers. If no line number is given, **ed** prints the current line, the line that **dot** refers to. If one line number is given (with or without the letter p), **ed** prints that line (and **dot** is set there); and if two line numbers are given, **ed** prints all the lines in that range (and sets **dot** to the last line printed). If two line numbers are specified, the first cannot be bigger than the second.

Pressing **CR** once causes printing of the next line. It is equivalent to:

```
.+1p
```

Next, try typing a minus sign (-) by itself; it is equivalent to typing:

```
.-1p
```

DELETING LINES: d

You use the **delete(d)** command to delete lines in the buffer. The lines to be deleted are specified for **d** exactly as they are for **p**. Thus, the command:

```
4,$d
```

deletes lines 4 through the end. There are now three lines left in our example, as you can check by typing:

```
1,$p
```

Notice that **$** now is line 3. **dot** is set to the line after the last line deleted, unless the last line deleted is the last line in the buffer. In that case, **dot** is set to **$.**

PERFORMING TEXT SUBSTITUTIONS: s

The **substitute(s)** command lets you change individual words or letters within a line or group of lines. It is the command you use to correct spelling mistakes and typing errors.

Suppose that, due to a typing error, line 1 says:

```
Now is th time
```

The letter "e" has been left off of the word "the". You can use **s** to correct this:

```
1s/th/the/
```

This substitutes the characters "the" for the characters "th" in line 1. To verify that the substitution has worked, type:

```
p
```

to get:

```
Now is the time
```

which is what you wanted. Notice that **dot** must be the line where the substitution took place, since the **p** command printed that line. **dot** is always set this way with the **s** command.

The syntax for the substitute command is:

[*starting-line* , *ending-line*]s/ *pattern* / *replacement* / *cmds*

The string of characters between the first pair of slashes is replaced by whatever is between the second pair, in all the lines between *starting-line* and *ending-line* . Only the first occurrence on each line is changed, however. Changing every occurrence is discussed later in this section. The rules for line numbers are the same as those for **p**, except that **dot** is set to the last line changed. If no substitution takes place, **dot** is not changed, resulting in the error message:

```
?
search string not found
```

Thus, you can type:

```
1,$s/speling/spelling/
```

and correct the first spelling mistake on each line in the text.

To change all occurrences of the text sequence, add a **g** (for global) to the **s** command:

```
s/ ... / ... /g
```

If no line numbers are given, the **s** command assumes we mean "make the substitution on line dot", so it changes things only on the current line. This leads to the very common sequence:

```
s/something/something else/p
```

which makes a correction on the current line, then prints it to make sure the correction worked out right. If it didn't, you can try again. (Notice that the **p** is on the same line as the **s** command. With few exceptions, **p** can follow any command; no other multicommand lines are legal.)

It is also legal to type:

```
s/string//
```

which means "change the first string of characters to nothing" or, in other words, remove them. This is useful for deleting extra words in a line or removing extra letters from words. For instance, if you had:

```
Nowxx is the time
```

you could type:

```
s/xx//p
```

to get:

```
Now is the time
```

Notice that two adjacent slashes mean "no characters", not a space.

There *is* a difference.

SEARCHING

Now that you've mastered the substitute command, you can move on to mastering another important concept: context searching.

Suppose you have the original three-line text in the buffer:

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

Suppose you want to find the line that contains the word "their", so that you can change it to the word "the". With only three lines in the buffer, it's pretty easy to keep track of which line the word "their" is on. But if the buffer contained several hundred lines, and you'd been making changes, deleting and rearranging lines, and so on, you would no longer really know what this line number would be. Context searching is simply a method of specifying the desired line, regardless of its number, by specifying a textual pattern contained in the line.

The way to say search for a line that contains this particular string of characters is to type:

```
/search_string/
```

For example, the **ed** command:

```
/their/
```

locates the next occurrence of the characters between the slashes (i.e., "their"). Note that you do not need to type the final slash. The above search command is the same as typing:

```
/their
```

The search command sets **dot** to the line on which the pattern is found and prints it for verification:

```
to come to the aid of their party.
```

"Next occurrence" means that **ed** starts looking for the string at line ".+1", searches to the end of the buffer, then continues at line 1 and searches to line dot. (That is, the search "wraps around" from $ to 1.) It scans all the lines in the buffer until it either finds the desired line or returns to **dot**. If the given string of characters can't be found in any line, **ed** prints the error message:

```
?
search string not found
```

Otherwise, **ed** prints the line it found.

You can also search backwards in a file for search strings by using question marks instead of slashes. For example:

```
?thing?
```

searches backwards in the file for the word "thing" as does:

```
?thing
```

The slash and question mark are the only characters you can use to delimit a search, though you can use any character in a substitute command. If you get unexpected results using any of the characters:

```
^   .   $   [   *   \   &
```

read the section entitled "Context and Regular Expressions".

You can search for the desired line and make a substitution at the same time by using a command of the form:

```
/their/s/their/the/p
```

This yields:

```
to come to the aid of the party.
```

The above command contains three separate actions:

- a context search for the desired line
- a substitution
- the printing of the line

The expression "/their/" is a context search expression. In their simplest form, all context search expressions are like this - a string of characters surrounded by delimiters. Context searches are interchangeable with line numbers, so they can be used by themselves to find and print a desired line, or as line numbers for some other command, like s. They were used both ways in the previous examples.

Suppose the buffer contains the three familiar lines:

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

The ed line numbers:

```
/Now/+1
/good/
/party/-1
```

are all context search expressions, and they all refer to the same line (line 2). To make a change in line 2, you could type:

```
/Now/+1s/good/bad/
```

or

```
/good/s/good/bad/
```

or

```
/party/-1s/good/bad/
```

The choice is dictated only by convenience. For instance, you could print all three lines by typing:

```
/Now/,/party/p
```

or

```
/Now/,/Now/+2p
```

or any similar combination. The first combination is better if you don't know how many lines are involved.

The basic rule is that a context search expression is the same as a line number, so it can be used wherever a line number is needed.

Suppose you search for:

```
/horrible thing/
```

and when the line is printed you discover that it isn't the "horrible thing" that you wanted, so it is necessary to repeat the search. You don't have to retype the search; just type:

```
//
```

This is a shorthand expression for "the previous thing that was searched for", whatever it was. This can be repeated as many times as necessary. You can also go backwards, since:

```
??
```

searches for the same thing, but in the reverse direction.

You can also use // as the left side of a substitute command, to mean "the most recent pattern." For example, if you've just found the string:

```
/horrible thing/
```

you can change it by typing:

```
s//good/p
```

This changes "horrible thing" to "good". To change previous occurrences of to "good", type:

```
??s//good/
```

CHANGING AND INSERTING TEXT: c AND i

This section discusses the **change**(c) command, which is used to change or replace one or more lines, and the **insert**(i) command, which is used for inserting one or more lines.

The **c** command is used to replace a number of lines with different lines that you type at the terminal. For example, to change lines ".+1" through "$" to something else, type:

```
.+1,$c
type the lines of text you want here ...
```

The lines you type between the **c** command and the **dot** (.) will replace the originally addressed lines. This is useful in replacing a line or several lines that have errors in them.

If only one line is specified in the **c** command, then only that line is replaced. (You can type in as many replacement lines as you like.) Notice the use of a period to end the input. This works just like the period in the **append** command and must appear by itself on a new line. If no line number is given, the current line specified by **dot** is replaced. The value of **dot** is set to the last line you typed in. Note that the line referenced by **dot** and the terminating period are completely different: the first is used simply to terminate a command, the second points at a specific line of text.

The **i** command is similar to the append command. For example:

```
/search string/i
type the lines to be inserted here ...
.
```

inserts the given text before the next line that contains "search string". The text between **i** and the terminating period is inserted before the specified line. If no line number is specified, **dot** is used. **dot** is set to the last line inserted.

MOVING LINES: m

The **move**(**m**) command lets you move a group of lines from one place to another in the buffer. Suppose you want to move the first three lines of the buffer to the end of the buffer. You could do it by typing:

```
1,3w temp
$r temp 1,3d
```

where *temp* is the name of a temporary file. However, you can do it more easily with the **m** command:

```
1,3m$
```

This will move lines 1 through 3 to the end of the file.

To move text, use a command of the form:

start-line , *end-line* m *after-this-line*

The third line specified is the place where the moved text is placed. Of course, the lines to be moved can be specified by context searches. If you had:

```
First paragraph
end of first paragraph.
Second paragraph
end of second paragraph.
```

you could reverse the two paragraphs like this:

```
/Second/,/end of second/m/First/-1
```

Notice the -1. The moved text goes after the line mentioned. **dot** gets set to the last line moved. Your file will now look like this:

```
Second paragraph
end of second paragraph
First paragraph
end of first paragraph
```

As another example of a frequent operation, you can reverse the order of two adjacent lines by moving the first line after the second line. Suppose that you are positioned at the first line. Then:

```
m+
```

moves line **dot** to one line after the current line **dot**. If you are positioned on the second line,

```
m--
```

moves line dot to one line after the current line **dot**.

The **m** command is more succinct than writing, deleting and rereading. The main difficulty with the **m** command is that if you use patterns to specify both the lines you are moving and the target, you must specify them properly, or you won't move the lines you want. The result of a bad **m** command can be a mess. Doing the job one step at a time makes it easier for you to verify at each step that you accomplished what you wanted. It is also a good idea to issue a **w** command before doing anything complicated; then if you make a mistake, you have the file stored in its original state so you can start again.

PERFORMING GLOBAL COMMANDS: g AND v

The "global" commands **g** and **v** are used to execute one or more editing commands on all lines that either contain (g) or don't contain (v) a specified pattern.

For example, the command:

```
g/XENIX/p
```

prints all lines that contain the word XENIX. The pattern that goes between the slashes can be anything that could be used in a line search or in a substitute command; exactly the same rules and limitations apply.

For example:

```
g/^\./p
```

prints all the **troff** formatting commands in a file (lines that begin with "."). (For an explanation of the use of the caret (^) and the backslash (\), see the section entitled "Context and Regular Expressions".)

The **v** command is identical to **g**, except that it operates on those lines that do not contain an occurrence of the pattern. (Mnemonically, the "v" can be thought of as part of the word "inverse".

For example:

```
v/^\./p
```

prints all the lines that don't begin with a period (i.e., the actual text lines).

Any command can follow **g** or **v**. For example, the following command deletes all lines that begin with ".":

```
g/^\./d
```

This command deletes all empty lines:

```
g/^$/d
```

Probably the most useful command that can follow a global command is the substitute command. For example, we could change the word "Xenix" to "XENIX" everywhere, and verify that it really worked, with:

```
g/Xenix/s//XENIX/gp
```

Notice that we used // in the substitute command to mean "the previous pattern," in this case, "Xenix". The **p** command executes on each line that matches the pattern, not just on those in which a substitution took place.

The global command makes two passes over the file. On the first pass, all lines that match the pattern are marked. On the second pass, each

marked line is examined in turn, **dot** is set to that line, and the command executed. This means that it is possible for the command that follows a **g** or **v** command to use addresses, set **dot**, and so on, quite freely. For example:

```
g/^\.P/+
```

prints the line that follows each **.P** command (the signal for a new paragraph in some formatting packages). Remember that plus (+) means "one line past dot." And:

```
g/topic/?^\.H?p
```

searches for each line that contains the word topic", scans backwards until it finds a line that begins with a **.H** (a heading) and prints it, thus showing the headings under which "topic" is mentioned. Finally:

```
g/^\.EQ/+,/^\.EN/-p
```

prints all the lines that lie between lines beginning with **.EQ** and **.EN** formatting commands.

The **g** and **v** commands can also be preceded by line numbers, in which case the lines searched are only those in the range specified.

It is possible to give more than one command under the control of a global command. For example, suppose the task is to change "x" to "y" and "a" to "b" on all lines that contain "thing". Then:

```
g/thing/s/x/y/\
s/a/b/
```

is sufficient. The backslash (\) signals the **g** command that the set of commands continues on the next line; the **g** command terminates on the first line that does not end with a backslash.

Note that you cannot use a substitute command to insert a new line within a **g** command.

The command:

```
g/x/s//y/\
s/a/b/
```

does *not* work as you might expect. The remembered pattern is the last pattern that was actually executed, so sometimes it will be "x" (as expected), and sometimes it will be "a" (not expected). You must spell it out, like this:

```
g/x/s/x/y/\
s/a/b/
```

It is also possible to execute **a**, **c** and **i** commands as part of a global command. As with other multiline constructions, add a backslash at the end of each line except the last. Thus, to add an **.nf** and **.sp** command

before each .EQ line, type:

```
g/^\.EQ/i\
.nf\
.sp
```

There is no need for a final line containing a period (.) to terminate the i command, unless there are further commands to be executed under the global command.

DISPLAYING TABS AND CONTROL CHARACTERS: l

ed provides two commands for printing the contents of the text you are editing. You should already be familiar with p, in combinations like:

```
1,$p
```

to print all the lines you are editing, or:

```
s/abc/def/p
```

to change "abc" to "def" on the current line. Less familiar is the **list** (l) command which gives slightly more information than p. In particular, l makes visible characters that are normally invisible, such as tabs and backspaces. If you list a line that contains some of these, l prints each tab as ">" and each backspace as "<". This makes it much easier to correct the sort of typing mistake that inserts extra spaces adjacent to tabs, or inserts a backspace followed by a space.

The l command also "folds" long lines for printing. Any line that exceeds 72 characters is printed on multiple lines; each printed line except the last is terminated by a backslash (\), so you can tell it was folded. This is useful for printing lines longer than the width of your terminal screen.

Occasionally, the l command will print a string of numbers preceded by a backslash, such as \07 or \16. These combinations are used to make visible characters that normally don't print, like form feed, vertical tab, or bell. Each backslash-number combination represents a single ASCII character. Note that numbers are octal and not decimal. When you see such characters, be wary: they may have surprising meanings when printed on some terminals. Often their presence indicates an error in typing, because they are rarely used.

UNDOING COMMANDS: u

Occasionally you will make a substitution in a line, only to realize too late that it was a mistake. The **undo(u)** command, lets you "undo" the last substitution. Thus the last line that was substituted can be restored to its previous state by typing:

```
u
```

This command does not work with the **g** and **v** commands.

MARKING YOUR SPOT IN A FILE: k

The **mark** command, **k**, provides a facility for marking a line with a particular name, so that you can later reference it by name, regardless of its actual line number. This can be handy for moving lines and keeping track of them as they move. For example:

```
kx
```

marks the current line with the name "x" . If a line number precedes the k, that line is marked. (The mark name must be a single lowercase letter.) You can refer to the marked line with the notation:

```
'x
```

Note the use of the single, close quotation mark (') here. Marks are very useful for moving things around. Find the first line of the block to be moved and then mark it with:

```
ka
```

Then find the last line and mark it with:

```
kb
```

Go to at the place where the text is to be inserted and type:

```
'a,'bm.
```

A line can have only one mark name associated with it at any given time.

TRANSFERRING LINES: t

We mentioned earlier the idea of saving lines that are hard to type or used often, to cut down on typing time. **ed** provides another command, called **t** (for **transfer**) for making a copy of a group of one or more lines at any point. This is often easier than writing and reading.

The **t** command is identical to the **m** command, except that instead of moving lines it simply duplicates them at the place you named. Thus:

```
1,$t$
```

duplicates the entire contents that you are editing.

A common use for **t** is to create a series of lines that differ only slightly. For example, you can type:

```
a
Now is the time for all good men to come to the aid of their party.
.
t.                          [make a copy]
s/men/women/                [change it a bit]
t.                          [make third copy]
s/Now is/yesterday was/     [change it a bit]
```

Your file will look like this:

Now is the time for all good men to come to the aid of their party.
Now is the time for all good women to come to the aid of their party.
Yesterday was the time for all good women to come to the aid of their party.

ESCAPING TO THE SHELL: !

Sometimes it is convenient to temporarily escape from the editor to execute a XENIX command. The shell escape (!) command, provides a way to do this.

If you type:

```
! command
```

your current editing state is suspended, and the XENIX command you asked for is executed. When the command finishes, **ed** will signal you by printing another exclamation mark (!); at that point you can resume editing.

CONTEXT AND REGULAR EXPRESSIONS

You may have noticed that things don't work right when you use characters such as the period (.), the asterisk (*), and the dollar sign ($) in context searches and with the substitute command. **ed** treats these characters as special. For instance, in a context search or the first string of the substitute command, the period (.) means any character, not a period, so:

```
/x.y/
```

means a line with an "x", any character, and a "y", not just a line with an "x", a period, and a "y".

The following characters are considered special:

^	caret
.	period
$	dollar sign
[ ]	brackets
*	asterisk
\	backslash
/	slash

The next few subsections discuss how to use these characters to describe regular expressions - patterns of text in search and substitute commands.

Recall that a trailing **g** after a substitute command causes all occurrences to be changed. With:

```
s/this/that/
```

and:

```
s/this/that/g
```

the first command replaces the first "this" on the line with "that." If there is more than one "this" on the line, the second form with the trailing **g** changes all of them.

Either form of the **s** command can be followed by **p** or **l** to print or list the contents of the line. For example, all of the following command lines are legal and mean slightly different things:

```
s/this/that/p
s/this/that/l
s/this/that/gp
s/this/that/gl
```

Make sure you know what the differences are.

Of course, any **s** command can be preceded by one or two line numbers to specify that the substitution is to take place on a group of lines. Thus:

```
1,$s/mispell/misspell/
```

changes the first occurrence of "mispell" to "misspell" in each line of the file. But:

```
1,$s/mispell/misspell/g
```

changes every occurrence in each line (and this is more likely to be what you wanted).

If you add a **p** or **l** to the end of any of these substitute commands, only the last line changed is printed, not all the lines. We will talk later about how to print all the lines that were modified.

PERIOD: (.)

On the left side of a substitute command, or in a search, a period stands for any single character. Thus the search:

```
/x.y/
```

finds any line where "x" and "y" occur separated by a single character, as in:

```
x+y
x-y
x y
xzy
```

Since a period matches a single character, it gives you a way to deal with funny characters printed by **l**. Suppose you have a line that appears as

```
th\07is
```

when printed with the **l** command, and that you want to get rid of the \07, which represents an ASCII bell character.

The most obvious solution is to try:

```
s/\07//
```

but this will fail. Another solution is to retype the entire line. This is guaranteed, and is actually quite a reasonable tactic if the line in question isn't too long. But for a very long line, retyping is not the best solution. This is where the metacharacter "." comes in handy. Since \07 really represents a single character, if we type:

```
s/th.is/this/
```

the job is done. The period matches the mysterious character between the "h" and the "i", whatever it is.

Since the period matches any single character, the command:

```
s/./,/
```

converts the first character on a line into a comma (,), which very often is not what you intended. The special meaning of the period can be removed by preceding it with a backslash.

As is true of many characters in ed, the period (.) has several meanings, depending on its context. This line shows all three:

```
.s/./.
```

The first period refers to the line number of the line we are editing, which is called **dot**. The second period refers to a metacharacter that matches any single character on that line. The third period is the only one that really is an honest, literal period. (Remember that a period is also used to terminate input from the **a** and **i** commands.) On the right side of a substitution, the period (.) is not special. If you apply this command to the line:

```
Now is the time.
```

the result is:

```
.ow is the time.
```

which is probably not what you intended. To change the period at the end of the sentence to a comma, type:

```
s/\./,/
```

The special meaning of the period can be removed by preceding it with a backslash. Use of the backslash is discussed in the next section.

BACKSLASH: \

Since a period means "any character", the question naturally arises: what do you do when you really want a period? For example, how do you convert the line:

```
Now is the time.
```

into:

```
Now is the time?
```

The backslash (\) turns off any special meaning that the next character might have; in particular, "\." converts the "." from a "match anything" into a literal period, so you can use it to replace the period in "Now is the time." like this:

```
s/\./?/
```

The pair of characters "\." is considered by **ed** to be a single real period.

The backslash can also be used when searching for lines that contain a special character. Suppose you are looking for a line that contains:

```
.DE
```

The search:

```
/.DE/
```

isn't adequate, for it will find lines like:

```
JADE
FADE
MADE
```

because the "." matches the letter "A" on each of the lines in question. But if you type:

```
/\.DE/
```

only lines that contain ".DE" are found.

The backslash can be used to turn off special meanings for characters other than the period. For example, consider finding a line that contains a backslash. The search:

```
/\/
```

won't work, because the backslash (\) isn't a literal backslash, but instead means that the second slash (/) no longer delimits the search. By preceding a backslash with another backslash, you can search for a literal backslash:

```
/\\/
```

You can search for a forward slash (/) with

```
/\//
```

The backslash turns off the special meaning of the slash immediately following so that it doesn't terminate the slash-slash construction prematurely. A miscellaneous note about backslashes and special characters: you can use any character to delimit the pieces of an s command; there is nothing sacred about slashes. (But you must use slashes for context searching.) For instance, in a line that contains several slashes already, such as:

```
//exec //sys.fort.go // etc...
```

you could use a colon as the delimiter. To delete all the slashes, type:

```
s:/::g
```

The result is:

```
exec sys.fort.go etc...
```

When you are adding text with **a** or **i** or **c**, the backslash has no special meaning, and you should put in only one backslash for each one you want.

DOLLAR SIGN: $

The dollar sign "$", stands for Suppose you have the line

```
Now is the
```

and you want to add the word "time" to the end. Use the dollar sign ($) like this:

```
s/$/ time/
```

to get:

```
Now is the time
```

A space is needed before "time" in the substitute command, or you will get:

```
Now is thetime
```

You can replace the second comma in the following line with a period without altering the first.

```
Now is the time, for all good men,
```

The command needed is:

```
s/,$/./
```

to get:

```
Now is the time, for all good men.
```

The dollar sign ($) here provides context to make specific which comma we mean. Without it the **s** command would operate on the first comma to produce:

```
Now is the time. for all good men,
```

To convert:

```
Now is the time.
```

into:

```
Now is the time?
```

as we did earlier, we can use:

```
s/.$/?/
```

Like the period (.), the dollar sign ($) has multiple meanings depending on context. In the following line:

```
$s/$/$/
```

the first "$" refers to the last line of the file, the second refers to the end of that line, and the third is a literal dollar sign to be added to that line.

CARET: ^

The caret (^) stands for the beginning of the line. For example, suppose you are looking for a line that begins with "the". If you simply type:

```
/the/
```

you will probably find several lines that contain "the" in the middle before arriving at the one you want. But with:

```
/^the/
```

you narrow the context, and thus arrive at the desired line more easily.

The other use of the caret (^) enables you to insert something at the beginning of a line. For example:

```
s/^/ /
```

places a space at the beginning of the current line.

Metacharacters can be combined. To search for a line that contains only the characters:

```
.P
```

you can use the command:

```
/^\.P$/
```

ASTERISK: *

Suppose you have a line that looks like this:

text x y *text*

where *text* stands for lots of text, and there are an indeterminate number of spaces between the "x" and the "y". Suppose the job is to replace all the spaces between "x" and "y" with a single space. The line is too long to retype, and there are too many spaces to count.

This is where the metacharacter "asterisk" (*) comes in handy. A character followed by a asterisk stands for as many consecutive occurrences of that character as possible. To refer to all the spaces at once, type:

```
s/x *y/x y/
```

The "*" means "as many spaces as possible." Thus "x *y" means an "x", as

many spaces as possible, then a "y".

The asterisk can be used with any character, not just a space. If the original example were:

text x--------y *text*

then all minus signs (-) can be replaced by a single space with the command:

s/x-*y/x y/

Finally, suppose that the line were:

text x.................y *text*

If you blindly type:

s/x.*y/x y/

the result is unpredictable. If there are no other x's or y's on the line, the substitution will work, but not necessarily. The period matches any single character so the ".*" matches as many single characters as possible, and unless you are careful, it can remove more of the line than you expected. For example, if the line were:

x *text* x.........y *text* y

then typing:

s/x.*y/x y/

takes everything from the first "x" to the last: "y", which, in this example, is undoubtedly more than you wanted.

The solution is to turn off the special meaning of the period (.) with the backslash (\):

s/x\.*y/x y/

Now the substitution works, for "\.*" means "as many periods as possible".

There are times when the pattern ".*" is exactly what you want. For example, to change:

Now is the time for all good men ....

into:

Now is the time.

use ".*" to remove everything after the "for":

s/ for.*/./

There are a couple of additional pitfalls associated with the asterisk (*). Most notable is the fact that "as many as possible" means zero or more. The fact that zero is a legitimate possibility is sometimes rather surprising. For example, if our line contained:

xy *text* x y *text*

and we said:

s/x *y/x y/

the first "xy" matches this pattern, for it consists of an zero spaces, and a "y". The result is that the substitute acts on the first "xy", and does not touch the later one that actually contains some intervening spaces.

The way around this is to specify a pattern like

/x *y/

which says an "x", a space, then as many more spaces as possible, then a "y", in other words, one or more spaces.

The other pitfall associated with the asterisk (*) again relates to the fact that zero is a legitimate number of occurrences of something followed by a asterisk. The command:

s/x*/y/g

when applied to the line:

abcdef

produces:

yaybycydyeyfy

which is almost certainly not what was intended. The reason for this is that zero is a legitimate number of matches, and there are no x's at the beginning of the line (so that gets converted into a "y"), nor between the "a" and the "b" (so that gets converted into a "y"), and so on. If you don't want zero matches, use:

s/xx*/y/g

since "xx*" is one or more x's.

BRACKETS: [AND]

Suppose that you want to delete any numbers that appear at the beginning of all lines of a file. You might try a series of commands like:

```
1,$s/^1*//
1,$s/^2*//
1,$s/^3*//
```

and so on, but this is clearly going to take forever if the numbers are at all long. Unless you want to repeat the commands over and over until finally all the numbers are gone, you must get all the digits on one pass. That is the purpose of the brackets.

The construction:

```
[0123456789]
```

matches any single digit-the whole thing is called a character class. With a character class, the job is easy. The pattern "[0123456789]*" matches zero or more digits (an entire number), so:

```
1,$s/^[0123456789]*//
```

deletes all digits from the beginning of all lines.

Any characters can appear within a character class, and there are only three special characters (^] and -) inside the brackets; even the backslash doesn't have a special meaning. To search for special characters, for example, you can type:

```
/[.\$^[]/
```

It's a nuisance to have to spell out the digits, so you can abbreviate them as [0-9]; similarly, [a-z] stands for the lowercase letters, and [A-Z] for uppercase.

Within [ ], the "[" is not special. To get a "]" (or a "-") into a character class, make it the first character.

You can also specify a class that means "none of the following characters." This is done by beginning the class with a caret (^). For example:

```
[^0-9]
```

stands for "any character exceptp a digit" . Thus, you might find the first line that doesn't begin with a digit with a search like:

```
/^[^0-9]/
```

Within a character class, the caret has a special meaning only if it occurs at the beginning. Just to convince yourself, verify that:

```
/^[^^]/
```

finds a line that doesn't begin with a caret.

AMPERSAND: &

To save typing, use the ampersand (&) to signify the string of text that was found on the left side of a substitute command. Suppose you have the line:

Now is the time

and you want to make it:

Now is the best time

You can type:

```
s/the/the best/
```

It's unnecessary to repeat the word "the". The ampersand (&) eliminates this repetition. On the right side of a substitution, the ampersand means "whatever was just matched", so you can type:

```
s/the/& best/
```

and tne ampersand will stand for "the". This isn't much of a saving if the thing matched is just "the", but if the match is very long, or if it is something like ".*" which matches a lot of text, you can save some tedious typing. There is also much less chance of making a typing error in the replacement text. For example, to put parentheses in a line, regardless of its length, type:

```
s/.*/(&)/
```

The ampersand can occur more than once on the right side. For example:

```
s/the/& best and & worst/
```

makes:

Now is the best and the worst time

and:

```
s/.*/&? &!!/
```

converts the original line into:

Now is the time? Now is the time!!

To get a literal ampersand use the backslash to turn off the special meaning. For example:

```
s/ampersand/\&/
```

converts the word into the symbol. The ampersand is not special on the left side of a substitute command, only on the right side.

SUBSTITUING NEW LINES

ed provides a facility for splitting a single line into two or more shorter lines by substituting in a newline. For example, suppose a line has become unmanageably long because of editing. If it looks like:

text xy *text*

you can break it between the "x" and the "y" like this:

```
s/xy/x\
y/
```

This is actually a single command, although it is typed on two lines. Because the backslash (\) turns off special meanings, a backslash at the end of a line makes the newline there no longer special.

You can in fact make a single line into several lines with this same mechanism. As an example, consider italicizing the word "very" in a long line by splitting "very" onto a separate line, and preceding it with the formatting command ".I". Assume the line in question looks like this:

text a very big text

The command:

```
s/ very /\
.I\
very\
/
```

converts the line into four shorter lines, preceding the word "very" with the command line .I , and eliminating the spaces around the "very" at the same time.

When a new line is substituted in a string, **dot** is left at the last line created.

JOINING LINES

Lines may be joined together, with the j command. Assume that you are given the lines

```
Now is
 the time
```

Suppose that **dot** is set to the first line. Then the command:

```
j
```

joins them together to produce:

```
Now is the time
```

No spaces are added, which is why a space was shown at the beginning of the second line.

All by itself, a **j** command joins the lines signified by **dot** and **dot + 1**, but any contiguous set of lines can be joined. Just specify the starting and ending line numbers. For example:

```
1,$jp
```

joins all the lines in a file into one big line and prints it.

REARRANGING A LINE: \(and \)

Recall that "&" is shorthand for whatever was matched by the left side of an **s** command. In much the same way, you can capture separate pieces of what was matched. The only difference is that you have to specify on the left side just what pieces you're interested in.

Suppose that you have a file of lines that consist of names in the form:

```
Smith, A. B.
Jones, C.
```

and so on, and you want the initials to precede the name, as in:

```
A. B. Smith
C. Jones
```

It is possible to do this with a series of editing commands, but it is tedious and error-prone.

The alternative is to "tag" the pieces of the pattern (in this case, the last name, and the initials), then rearrange the pieces. On the left side of a substitution, if part of the pattern is enclosed between \(and \), whatever matched that part is remembered, and available for use on the right side. On the right side, the symbol "\1" refers to whatever matched the first \(...\) pair; "\2", to the second \(...\), and so on.

The command:

```
1,$s/^\([.*]\), *\(.*\)/\2 \1/
```

although hard to read, does the job. The first \(...\) matches the last name, which is any string up to the comma; this is referred to on the right side with "\1". The second \(...\) is whatever follows the comma and any spaces, and is referred to as "\2".

With any editing sequence this complicated, it's unwise to simply run it and hope. The global commands **g** and **v** provide a way for you to print exactly those lines which were affected by the substitute command, and

thus verify that it did what you wanted in all cases.

SPEEDING UP EDITING

One of the most effective ways to speed up your editing is knowing what lines will be affected by a command if you don't specify the lines it is to act on, and on what line you will be positioned (i.e., the value of dot) when a command finishes. If you can edit without specifying unnecessary line numbers, you can save a lot of typing.

For example, if you issue a search command like:

```
/thing/
```

you are left pointing at the next line that contains "thing". Then no address is required with commands like **s** to make a substitution on that line, or **p** to print it, or **l** to list it, or **d** to delete it, or **a** to append text after it, or **c** to change it, or **i** to insert text before it.

What happens if there is no occurrence of "thing" ? **dot** is unchanged. This is also true if the cursor was on the only occurrence of "thing" when you issued the command. The same rules hold for searches that use ?...?; the only difference is the direction in which you search.

The delete command, **d**, leaves **dot** pointing at the line that followed the last deleted line. When the line dollar ($) gets deleted, however, **dot** points at the new line $.

The line-changing commands **a** ,**c** , and **i** , by default, all affect the current line. If you give no line number with them, **a** appends text after the current line, **c** changes the current line, and **i** inserts text before the current line.

The **a** , **c** , and **i** commands behave identically in one respect - when you stop appending, changing or inserting, **dot** points at the last line entered. This is exactly what you want when typing and editing on the fly. For example, you can type:

```
a
text
botch   (minor error)
.
s/botch/correct/  (fix botched line)
a
more text
.
```

without specifying any line number for the substitute command or for the second append command. Or you can type:

```
a
text
horrible botch  (major error)
.
c       (replace entire line)
fixed up line
.
```

Experiment to determine what happens if you add no lines with an **a** , **c** , or **i** command.

The **r** command reads a file into the text being edited, at the end if you give no address, or after the specified line if you do. In either case, **dot** points at the last line read in. Remember that you can even type:

```
0r
```

to read a file in at the beginning of the text. (You can also type 0a or 1i to start adding text at the beginning.)

The **w** command writes out the entire file. If you precede the command by one line number, that line is written out. If you precede it by two line numbers, that range of lines is written out. The **w** command does <not> change **dot** : the current line remains the same, regardless of what lines are written out. This is true even if you type something like:

```
/^\.AB/,/^\.AE/w abstract
```

which involves a context search.

(Since the **w** command is so easy to use, you should save what you are editing regularly as you go along just in case the system crashes, or in case you accidentally delete what you're editing.)

The general rule is simple: you are left sitting on the last line changed; if there were no changes, then **dot** is unchanged. To illustrate, suppose there are three lines in the buffer, and the line given by **dot** is the middle one:

```
x1
x2
x3
```

Then the command:

```
-,+s/x/y/p
```

prints the third line, which is the last one changed. But if the three lines had been:

```
x1
y2
y3
```

and the same command had been issued while **dot** pointed at the second

line, only the first line would be changed and printed, and that is where **dot** would be set.

SEMICOLON: ;

Searches with /.../ and ?...? start at the current line and move forward or backward, respectively, until they either find the pattern or return to the current line. Sometimes this is not what you want. Suppose, for example, that the buffer contains lines like this:

```
      .
      .
      .
   ab
      .
      .
      .
   bc
      .
      .
      .
```

Starting at line 1, you would expect the command:

```
/a/,/b/p
```

to print all the lines from the "ab" to the "bc" inclusive. This is not what happens. Both searches (for "a" and for "b") start from the same point, and thus they both find the line that contains "ab". As a result, a single line is printed. Worse, if there had been a line with a "b" in it before the "ab" line, then the print command would be in error, since the second line number would be less than the first, and it is illegal to try to print lines in reverse order.

This is because the comma separator for line numbers doesn't set **dot** as each address is processed; each search starts from the same place. In **ed**, the semicolon (;) can be used just like the comma, with the single difference that use of a semicolon forces **dot** to be set at the time the semicolon is encountered, as the line numbers are being evaluated. In effect, the semicolon "moves" **dot**. Thus, in our example above, the command:

```
/a/;/b/p
```

prints the range of lines from "ab" to "bc", because after the "a" is found, **dot** is set to that line, and then "b" is searched for, starting beyond that line.

This property is most often useful in a very simple situation. Suppose you want to find the second occurrence of "thing". You could type:

```
/thing/
//
```

but this prints the first occurrence as well as the second, and is a nuisance when you know very well that it is only the second one you're interested in. The solution is to type:

```
/thing/;//
```

This says "find the first occurrence of "thing", set **dot** to that line, then find the second occurrence and print only that."

Closely related is searching for the second to last occurrence of something, as in:

```
?something?;??
```

Finally, bear in mind that if you want to find the first occurrence of something in a file, starting at an arbitrary place within the file, it is not sufficient to type:

```
1;/thing/
```

because if "thing" occurs on line 1 it won't be found. The command:

```
0;/thing/
```

will work because it starts the search at line 1. This is one of the few places where 0 is a legal line number.

INTERRUPTING THE EDITOR

As a final note on what **dot** gets set to, you should be aware that if you press the **INTERRUPT** key while **ed** is executing a command, your file is restored, as much as possible, to what it was before the command began. Naturally, some changes are irrevocable - if you are reading in or writing out a file, making substitutions, or deleting lines. These will be stopped in some unpredictable state in the middle (which is why it usually unwise to stop them). **dot** may or may not be changed.

If you are using the print command, **dot** is not changed until the printing is done. Thus, if you decide to print until you see an interesting line, and then press **INTERRUPT** , to stop the command, **dot** will not be set to that line or even near it. **dot** is left where it was when the **p** command was started.

CUTTING AND PASTING WITH THE EDITOR

This section describes how to manipulate pieces of files, individual lines or groups of lines.

INSERTING ONE FILE INTO ANOTHER

Suppose you have a file called memo, and you want the file called table to be inserted just after a reference to Table 1. That is, in memo somewhere is a line that says:

```
Table 1 shows that ...
```

and the data contained in table has to go there.

To put table into the correct place in the file edit memo, find "Table 1", and add the file table right there:

```
ed memo
/Table 1/
response from ed
.r  table
```

The critical line is the last one. The **r** command reads a file; here you asked for it to be read in right after line **dot**. An **r** command, without any address, adds lines at the end, so it is the same as "$r".

WRITING OUT PART OF A FILE

The other side of the coin is writing out part of the document you're editing. For example, you may want to split the table from the previous example out into a separate file so it can be formatted and tested separately. Suppose that in the file being edited we have:

```
[lots of stuff]
```

which is the way a table is set up for the **tbl** program. To isolate the table in a separate file called table, first find the start of the table (the .TS line), then write out the interesting part. For example, first type:

```
/^\.TS/
```

This prints out the found line:

```
.TS
```

Next type:

```
.,/^\.TE/w table
```

and the job is done. If you are confident, you can do it all at once with:

```
/^\.TS/;/^\.TE/w table
```

The point is that the **w** command can write out a group of lines, instead of the whole file. In fact, you can write out a single line if you like; just give one line number instead of two. If you have just typed a

horribly complicated line and you know that it (or something like it) is going to be needed later, then save it - don't retype it. For example, in the editor, type:

```
a
lots of stuff
horrible line
.
.w  temp
a
more stuff
.
.r temp
a
more stuff
.
```

EDITING SCRIPTS

If a fairly complicated set of editing operations is to be performed on a whole set of files, the easiest thing to do is to make up a script , i.e., a file that contains the operations you want to perform, then apply this script to each file in turn.

For example, suppose you want to change every "Xenix" to "XENIX" and every "USA" to "America" in a large number of files. Put the following lines into the file script :

```
g/Xenix/s//XENIX/g
g/USA/s//America/g
w
q
```

Now you can type:

```
ed - file1 <script
ed - file2 <script
...
```

This causes **ed** to take its commands from the prepared file script. Notice that the whole job has to be planned in advance, and that by using the XENIX shell command interpreter, you can cycle through a set of files automatically. The dash (-) suppresses unwanted messages from **ed**.

When preparing editing scripts, you may need to place a period as the only character on a line to indicate termination of input from an **a** or **i** command. This is difficult to do in **ed** , because the period you type will terminate input rather than be inserted in the file. Using a backslash to escape the period won't work either. One solution is to create the script using a character such as the at-sign (@) to indicate end of input. Then, later, use the following command to replace the at-sign with a period:

```
s/^@$/./
```

SUMMARY OF COMMANDS

This following is a list of all **ed** commands. The general form of **ed** commands is the command name, preceded by one or two optional line numbers and, in the case of **e**, **f**, **r**, and **w** , followed by a filename. Only one command is allowed per line, but a **p** command may follow any other command (except **e**, **f**, **r**, **w**, and **q**).

a Appends, i.e., adds lines to the buffer (at line **dot** , unless a different line is specified). Appending continues until a period is typed on a new line. The value of **dot** is set to the last line appended.

c Changes the specified lines to the new text which follows. The new lines are terminated by a period on a new line, as with **a**. If no lines are specified, replace line **dot**. **dot** is set to the last line changed.

d Deletes the lines specified. If none are specified, deletes line **dot**. **dot** is set to the first undeleted line following the deleted lines unless dollar ($) is deleted, in which case dot is set to dollar.

e Edits a new file. Any previous contents of the buffer are thrown away, so issue a **w** command first.

f Prints the remembered filename. If a name follows **f** , then the remembered name is set to it.

g The command g */string/commands* executes *commands* on those lines that contain *string* , which can be any context search expression.

i Inserts lines before specified line (or **dot**) until a single period is typed on a new line. **dot** is set to the last line inserted.

l Lists lines, making visible nonprinting ASCII characters and tabs. Otherwise similar to **p**.

m Moves lines specified to after the line named after **m**. **dot** is set to the last line moved.

p Prints specified lines. If none are specified, print the line specified by **dot**. A single line number is equivalent to a line-number (**p**) command. A single **CR** prints ".+1", the next line.

q Quits **ed**. Your work is not saved unless you first use the **w** command. Press **q** twice to terminate an edit.

r Reads a file into buffer (at end unless specified elsewhere.) **dot** is set to the last line read.

s	The command s */string1/string2/* substitutes the pattern matched by *string1* with the string specified by *string2* in the specified lines. If no lines are specified, the substitution takes place only on the line specified by **dot**. **dot** is set to the last line in which a substitution took place, which means that if no substitution takes place, **dot** remains unchanged. The s command changes only the first occurrence of *string1* on a line; to change multiple occurrences on a line, type a **g** after the final slash.
t	Transfers specified lines to the line named after **t**. **dot** is set to the last line moved.
v	The command v */string/commands* executes commands on those lines that do not contain *string* .
u	Undoes the last substitute command.
w	Writes out the editing buffer to a file. **dot** remains unchanged.
.=	Prints value of **dot**. (An equal sign by itself prints the value of $.)
! *command*	The line ! *cmd-line* causes *cmd-line* to be executed as a XENIX command.
/string/	Context search. Searches for next line which contains this string of characters and prints it. **dot** is set to the line where *string* was found. The search starts at .+1, wraps around from $ to 1, and continues to **dot** , if necessary.
? *string* ?	Context search in reverse direction. Starts search at .-1, scans to 1, wraps around to $.

12. EDITING WITH sed AND awk

ABOUT THIS CHAPTER

This chapter serves as an introduction to the XENIX V noninteractive editors, **sed** and **awk** .

CONTENTS

INTRODUCTION

This chapter describes two XENIX utilities that allow you to perform large-scale, noninteractive editing tasks:

- **sed** a noninteractive, or batch, editor which is useful to work with large files or run a complicated sequence of editing commands on a file or group of files.

- **awk** , which searches numerics, logical relations, variables, and particular fields within lines of text.

Although you can perform many of the same tasks with **grep** , **sort** , and the variants of **diff** , you will find that these two programs offer an added facility for the processing of complicated changes to large files, or many files at once. **sed** is very handy for large batch editing jobs, but if you choose not to learn it, many of the same tasks can be performed with **ed** scripts. The **awk** program offers several features not available with the other tools described in this chapter, but it is somewhat more complicated to learn and use.

EDITING WITH sed

The **sed** program is a noninteractive editor which is especially useful when the files to be edited are too large, or the sequence of editing commands too complex, to be executed interactively. **sed** works on only a few lines of input at a time and does not use temporary files, so the only limit on the size of the files you can process is that both the input and output must be able to fit simultaneously on your disk. You can apply multiple global editing functions to your text in one pass. Since you can create complicated editing scripts and submit them to **sed** as a command file, you can save yourself considerable retyping and errors. You can also save and reuse **sed** command files which perform editing operations you need to repeat frequently.

Processing files with **sed** command files is more efficient than using **ed** , even if you prepare a prewritten script. Note, however, that **sed** lacks relative addressing because it processes a file one line at a time. Also, **sed** gives you no immediate verification that a command has altered your text in the way you actually intended. Check your output carefully.

The **sed** program is derived from **ed** , although there are considerable differences between the two that result from the different characteristics of interactive and batch operation. You will notice a striking resemblance in the class of regular expressions they recognize; the code for matching patterns is nearly identical for **ed** and **sed** .

OVERALL OPERATION

By default, **sed** copies the standard input to the standard output, performing one or more editing commands on each line before writing it to the output. Typically, you will need to specify the file or files you are processing, along with the name of the command file which contains your editing script, as in the following:

```
sed -f script filename
```

The general format of a **sed** editing command is:

```
address1,address2 function arguments
```

Addresses, functions and arguments are discussed in the following sections. In any command, one or both addresses may be omitted. A function is always required, but an argument is optional for some functions. Any number of **BLANKS** or **TABS** may separate the addresses from the function, and **TAB** characters and **SPACES** at the beginning of lines are ignored.

Three flags are recognized on the command line:

-n Directs **sed** to copy only those lines specified by **p** functions or **p** flags after **s** functions.

-e Indicates that the next argument is an editing command.

-f Indicates that the next argument is the name of the file which contains editing commands, typed one to a line.

All flags are optional. **sed** commands are applied one at a time, generally in the order they are encountered, unless you change this order with one of the flow-of-control functions discussed below. **sed** works in two phases, compiling the editing commands in the order they are given, then processing the input file using one command at a time.

The input to each command is the output of all preceding commands. Even if you change this default order of applying commands with one of the two flow-of-control commands, **t** and **b** , the input line to any command is still the output of any previously applied command.

You should also note that the range of pattern match is normally one line of input text. This range is called the pattern space. More than one line can be read into the pattern space by using the **N** command described in the section entitled "Functions", below.

The rest of this section discusses the principles of **sed** addressing, followed by a description of **sed** functions. All the examples here are based on the following lines from Samuel Taylor Coleridge's poem, "Kubla Khan":

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

For example, the command:

```
2q
```

will quit after copying the first two lines of the input. Using the sample text, the result will be:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
```

ADDRESSES

The following rules apply to addressing in **sed** . There are two ways to select the lines in the input file to which editing commands are to be applied: with line numbers or with context addresses. Context addresses correspond to regular expressions. The application of a group of commands can be controlled by one address or an address pair, by grouping the commands with curly braces ({ }). There may be 0, 1, or 2 addresses specified, depending on the command. The maximum number of addresses possible for each command is indicated.

A line number is a decimal integer. As each line is read from the input file, a line number counter is incremented. A line number address matches the input line, causing the internal counter to equal the address line number. The counter runs cumulatively through multiple input files; it is not reset when a new input file is opened. A special case is the dollar sign character ($) which matches the last line of the last input file.

Context addresses are enclosed in slashes (/). They include all the regular expressions common to both **ed** and **sed** :

- An ordinary character is a regular expression and matches itself.
- A caret (^) at the beginning of a regular expression matches the **NULL** character at the beginning of a line.
- A dollar sign ($) at the end of a regular expression matches the **NULL** character at the end of a line.
- The characters \n match an embedded **NEWLINE** character, but not the **NEWLINE** at the end of a pattern space.
- A period (.) matches any character except the terminal **NEWLINE** of the pattern space.
- A regular expression followed by a star (*) matches any number, including 0, of adjacent occurrences of the regular expression it

follows.

- A string of characters in square brackets ([]) matches any character in the string, and no others. If, however, the first character of the string is a caret (^), the regular expression matches any character except the characters in the string and the terminal **NEWLINE** of the pattern space.

- A concatenation of regular expressions is a regular expression which matches the concatenation of strings matched by the components of the regular expression.

- A regular expression between the sequences \(and \) is identical in effect to itself, but has side-effects with the **s** command.

- The expression **d** means the same string of characters matched by an expression enclosed in \(and \) earlier in the same pattern. Here Gr "d" is a single digit; the string specified is that beginning with the **d** th occurrence of \(counting from the left. For example, the expression ^\(.*\)\1 matches a line beginning with two repeated occurrences of the same string.

- The null regular expression standing alone is equivalent to the last regular expression compiled.

For a context address to match the input, the whole pattern within the address must match some portion of the pattern space. If you want to use one of the special characters literally, that is, to match an occurrence of itself in the input file, precede the character with a backslash (\) in the command.

Each **sed** command can have zero, one, or two addresses. The maximum number of allowed addresses is included. A command with no addresses specified is applied to every line in the input. If a command has one address, it is applied to all lines which match that address. On the other hand, if two addresses are specified, the command is applied to the first line which matches the first address, and to all subsequent lines until and including the first subsequent line which matches the second address. An attempt is made on subsequent lines to again match the first address, and the process is repeated. Two addresses are separated by a comma.

Here are some examples of **sed** commands:

/an/ Matches lines 1, 3, 4 in our "Kubla Khan" sample text

/an.*an/ Matches line 1

/^an/ Matches no lines

/./ Matches all lines

/r*an/ Matches lines 1,3, 4 (number = zero!)

FUNCTIONS

All sed functions are named by a single character. They are of the following types:

- Whole-line oriented functions, which add, delete, and change whole text lines.
- Substitute functions, which search for and substitute regular expressions within a line.
- Input-output functions, which read and write lines and/or files.
- Multiple input-line functions, which match patterns that extend across line boundaries.
- Hold and get functions, which save and retrieve input text for later use.
- Flow-of-control functions, which control the order of application of functions.
- Miscellaneous functions.

Whole-Line Oriented Functions

Whole-line oriented functions include the following:

d Deletes from the file all lines matched by its addresses. No further commands will be executed on a deleted line. As soon as the **d** function is executed, a new line is read from the input, and the list of editing commands is restarted from the beginning on the new line. The maximum number of addresses is two.

n Reads and replaces the current line from the input, writing the current line to the output if specified. The list of editing commands is continued following the **n** command. The maximum number of addresses is two.

a Causes the text to be written to the output after the line matched by its address. The **a** command is inherently multiline; **a** must appear at the end of a line. The text may contain any number of lines. The interior **NEWLINES** must be hidden by a backslash character (\) immediately preceding each **NEWLINE** . The text argument is terminated by the first unhidden **NEWLINE** , the first one not immediately preceded by backslash. Once an **a** function is successfully executed, the text will be written to the output regardless of what later commands do to the line which triggered it, even if the line is subsequently deleted. The text is not scanned for address matches, and no editing commands are attempted on it, nor does it cause any change in the line-number counter. Only one address is possible.

i When followed by a text argument, it is the same as the **a** function, except that the text is written to the output before the matched line. It has only one possible address. The **c** function deletes the lines selected by its addresses, and replaces them with the lines in the text. Like the **a** and **i** commands, **c** must be followed by a **NEWLINE** hidden with a backslash; interior **NEWLINES** in the text must be hidden by backslashes. The **c** command may have two addresses, and therefore select a range of lines. If it does, all the lines in the range are deleted, but only one copy of the text is written to the output, not one copy per line deleted. As in the case of **a** and **i**, the text is not scanned for address matches, and no editing commands are attempted on it. It does not change the line-number counter. After a line has been deleted by a **c** function, no further commands are attempted on it. If text is appended after a line by **a** or **r** functions, and the line is subsequently changed, the text inserted by the **c** function will be placed before the text of the **a** or **r** functions.

Note that when you insert text in the output with these functions, leading **BLANKS** and **TABS** will disappear in all **sed** commands. To get leading **BLANKS** and **TABS** into the output, precede the first desired **BLANK** or **TAB** by a backslash; the backslash will not appear in the output.

For example, the list of editing commands:

```
n
a\XXXX
d
```

applied to our standard input, produces:

```
In Xanadu did Kubla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
```

In this particular case, the same effect would be produced by either of the two following command lists:

```
n
i\
XXXX
d
```

or

```
n
c\
XXXX
```

Substitute Functions

The substitute function(s) changes parts of lines selected by a context search within the line, as in:

(2)s pattern replacement flags filename

The **s** function replaces part of a line selected by the designated pattern with the replacement pattern. The pattern argument contains a pattern, exactly like the patterns in addresses. The only difference between a pattern and a context address is that a pattern argument may be delimited by any character other than **SPACE** or **NEWLINE** . By default, only the first string matched by the pattern is replaced, except when the **-g** option is used.

The replacement argument begins immediately after the second delimiting character of the pattern, and must be followed immediately by another instance of the delimiting character. The replacement is not a pattern, and the characters which are special in patterns do not have special meaning in replacement. Instead, the following characters are special:

. The dot is replaced by the string matched by the pattern.

\d The letter **d** represents a single digit which is replaced by the **d** th substring matched by parts of the pattern enclosed in \ and \ . If nested substrings occur in the pattern, the **d** th substring is determined by counting opening delimiters.

As in patterns, special characters may be made literal by preceding them with a backslash (\).

A flag argument may contain the following:

g Substitutes the replacement for all nonoverlapping instances of the pattern in the line. After a successful substitution, the scan for the next instance of the pattern begins just after the end of the inserted characters; characters put into the line from the replacement are not rescanned.

p Prints the line if a successful replacement was done. The **p** flag causes the line to be written to the output if and only if a substitution was actually made by the **s** function. Notice that if several **s** functions, each followed by a **p** flag, successfully substitute in the same input line, multiple copies of the line will be written to the output: one for each successful substitution.

w file Writes the line to a file if a successful replacement was done. The **-w** option causes lines which are actually substituted by the **s** function to be written to the named file. If the filename existed before **sed** is run, it is overwritten; if not, the file is created. A single space must separate **-w** and the filename. The possibilities of multiple, somewhat different copies of one input line being written are the same as for the **-p** option. A combined maximum of ten different filenames may be mentioned

after **w** flags and **w** functions.

Here are some examples. When applied to our standard input, the following command:

```
s/to/by/w changes
```

produces, on the standard output:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless by man
Down by a sunless sea.
```

and on the file changes:

```
Through caverns measureless by man
Down by a sunless sea.
```

As a second example, here is the following **sed** command:

```
s/[.,;?:]/*P&*/gp
```

It produces the following text:

```
A stately pleasure dome decree*P:*
Where Alph*P,* the sacred river*P,* ran
Down to a sunless sea*P.*
```

With the **g** flag, the command:

```
/X/s/an/AN/p
```

produces:

```
In Xanadu did Kubla Khan
```

and the command:

```
/X/s/an/AN/gp
```

produces:

```
In XANadu did Kubla KhAN
```

Input-Output Functions

The input-output functions include:

p The print function writes the addressed lines to the standard output file at the time the **p** function is encountered, regardless of what succeeding editing commands may do to the lines. The maximum number of

possible addresses is two.

w filename The write function writes the addressed lines to filename. If the file previously existed, it is overwritten; if not, it is created. The lines are written exactly as they exist when the write function is encountered for each line, regardless of what subsequent editing commands may do to them. Exactly one space must separate the **w** command and the filename. The combined number of write functions and **w** flags may not exceed ten.

r The read function reads the contents of the named file, and appends them after the line matched by the address. The file is read and appended regardless of what subsequent editing commands do to the line which matched its address. If **r** and **a** functions are executed on the same line, the text from the **a** functions and the **r** functions is written to the output in the order that the functions are executed. Exactly one space must separate the **r** and the filename. One address is possible. If a file mentioned by an **r** function cannot be opened, it is considered a null file rather than an error, and no diagnostic is given.

Note that there is a limit to the number of files that can be opened simultaneously. Be sure that no more than ten files are mentioned in functions or flags; that number is reduced by one if any **r** functions are present. Only one read file is open at one time.

Here are some examples. Assume that the file **notel** has the following contents:

```
Note:  Kubla Khan (more properly Kublai Khan;
1216-1294) was the grandson and most eminent
successor of Genghiz (Chingiz) Khan, and
founder of the Mongol dynasty in China.
```

The following command:

```
/Kubla/r notel
```

produces:

```
In Xanadu did Kubla Khan
    Note:  Kubla Khan (more properly Kublai Khan;
    1216-1294) was the grandson and most eminent
    successor of Genghiz (Chingiz) Khan, and
    founder of the Mongol dynasty in China.
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

Multiple Input-Line Functions

Three functions, all spelled with uppercase letters, deal specially with pattern spaces containing embedded **NEWLINES** . They are intended principally to provide pattern matches across lines in the input.

N — Appends the next input line to the current line in the pattern space; the two input lines are separated by an embedded **NEWLINE** . Pattern matches may extend across the embedded **NEWLINE(S)** . There is a maximum of two addresses.

D — Deletes up to and including the first **NEWLINE** character in the current pattern space. If the pattern space becomes empty (the only **NEWLINE** was the terminal **NEWLINE**), another line is read from the input. In any case, begin the list of editing commands again from its beginning. The maximum number of addresses is two.

P — Prints up to and including the first **NEWLINE** in the pattern space. The maximum number of addresses is two.

The P and D functions are equivalent to their lowercase counterparts if there are no embedded newlines in the pattern space.

Hold and Get Functions

These functions save and retrieve part of the input for possible later use:

h — The h function copies the contents of the pattern space into a holding area, destroying any previous contents of the holding area. The maximum number of addresses is two.

H — The H function appends the contents of the pattern space to the contents of the holding area. The former and new contents are separated by a **NEWLINE** .

g — The g function copies the contents of the holding area into the pattern space, destroying the previous contents of the pattern space.

G — The G function appends the contents of the holding area to the contents of the pattern space. The former and new contents are separated by a **NEWLINE** . The maximum number of addresses is two.

x — The exchange command interchanges the contents of the pattern space and the holding area. The maximum number of addresses is two.

For example, the commands:

```
lh
ls/ did.*//
lx
G
s/0  :/
```

applied to our standard example, produce:

```
In Xanadu did Kubla Khan  :In Xanadu
A stately pleasure dome decree:  :In Xanadu
Where Alph, the sacred river, ran  :In Xanadu
Through caverns measureless to man  :In Xanadu
Down to a sunless sea.  :In Xanadu
```

Flow-of-Control Functions

These functions do no editing on the input lines, but control the application of functions to the lines selected by the address part.

! This command causes the next command written on the same line to be applied to only those input lines not selected by the address part. There are two possible addresses.

{ This command causes the next set of commands to be applied or not applied as a block to the input lines selected by the addresses of the grouping command. qThe first of the commands under control of the grouping command may appear on the same line as the **{** or on the next line. The group of commands is terminated by a matching **}** on a line by itself. Groups can be nested and may have two addresses.

:label The **label** function marks a place in the list of editing commands which may be referred to by **b** and **t** functions. The label may be any sequence of eight or fewer characters; if two different colon functions have identical labels, an error message will be generated, and no execution attempted.

blabel The branch function causes the sequence of editing commands being applied to the current input line to be restarted immediately after encountering a colon function with the same label. If no colon function with the same label can be found after all the editing commands have been compiled, an error message is produced, and no execution is attempted. A **b** function with no label is interpreted as a branch to the end of the list of editing commands. Whatever should be done with the current input line is done, and another input line is read; the list of editing commands is restarted from the beginning on the new line. Two addresses are possible.

tlabel The **t** function tests whether any successful substitutions have been made on the current input line. If so, it branches to the label; if not, it does nothing. The flag which indicates that a successful substitution has been executed is reset either by reading a new input line, or executing a **t** function.

Miscellaneous Functions

There are two other functions of **sed** not discussed above.

= The = function writes to the standard output the line number of the line matched by its address. One address is possible.

q The q function causes the current line to be written to the output (if it should be), any appended or read text to be written, and execution to be terminated. One address is possible.

PATTERN MATCHING WITH awk

By now you have been introduced to several tools for locating patterns and strings in one or more text files, including **grep** and its variants. You should also be familiar with using the various text editors to do global searching. **awk** offers another approach to many of these same tasks. **awk** is actually a programming language designed to make many common search and text manipulation tasks easy to state and to perform. It offers several key features not available with **grep** or **sed** : numeric processing, the handling of variables, general selection, and flow-of-control in commands. **awk** is also uniquely suited to operations on fields within lines.

In practice, **awk** is used in two ways: for report generation, processing input to extract counts, sums, subtotals, etc.; and to transform data from the form produced by one program into that expected by another. **awk** searches input lines consecutively for a match of any patterns which you designate. For each pattern, an action can be specified; this action will be performed on each line that matches the pattern. **awk** allows you to perform more complex actions than merely printing a matching line. For example, the **awk** program:

```
{print $3, $2}
```

prints the third and second columns of a table in that order. The program:

```
$2 ~/A|B|C/
```

prints all input lines with an A, B, or C in the second field, where the second field is text separated by whitespace. The program:

```
$1 != prev { print; prev = $1 }
```

prints all lines in which the first field is different from what was previously the first field.

STARTING awk

The command in the following form:

```
awk program filename
```

executes the **awk** commands written into the named program on the set of named files, or on the standard input if no files are named. The statements can also be placed in a file **pfile** , and executed by the command:

```
awk -f pfile filename
```

PROGRAM STRUCTURE

An **awk** program is a sequence of statements, each in the form:

```
pattern { action }
...
```

Each line of input is matched in turn against each of the specified patterns. For each pattern matched, the associated action is executed. When all the patterns have been tested, the next line is read and the matching process repeated. Either the pattern or the action may be omitted, but not both. If there is no action for a pattern, the matching line is simply copied to the output. Thus a line which matches several patterns can be printed several times. If there is no pattern for an action, then the action is performed for every input line. A line which matches no pattern is ignored. Since patterns and actions are both optional, actions must be enclosed in braces to distinguish them from patterns.

RECORDS AND FIELDS

awk input is divided into records which are terminated by a record separator. Because the default record separator is a **NEWLINE** , **awk** processes its input one line at a time. The number of the current record is available in a predefined variable named **NR** , for number register.

Each input record is divided into fields. Fields are normally separated by whitespace, either **SPACES** or **TABS** , but the input field separator can be changed. Fields are referred to as **$1** , **$2** , and so forth, where **$1** is the first field, and **$0** is the whole input record itself. Assignments may be made to fields. The number of fields in the current record is available in another predefined variable named **NF** , for number fields.

The variables FS and RS refer to the input field and record separators; they may be changed at any time to any single character. The optional command-line argument -Fc may also be used to set FS to the character c . If the record separator is empty, an empty input line is taken as the record separator, and blanks, tabs and newlines are treated as field separators. The variable filename contains the name of the current input file.

PRINTING

If an action has no pattern, the action is executed for all lines. The simplest action is to print some or all of a record, using the **awk** command print. This command prints each record, copying the input to the output intact. A field or group of fields may be printed from each record. For instance:

```
print $2, $1
```

prints the first two fields in reverse order. Items separated by a comma in the print statement will be separated by the current output field separator when output. Items not separated by commas will be concatenated by the print command. The following example runs the first and second fields together:

```
print $1 $2
```

The predefined variables **NF** and **NR** can be used. The following example prints each record preceded by the record number and the number of fields:

```
{ print NR, NF, $0 }
```

Output may be diverted to multiple files. For example, the program:

```
{ print $1 >"list1"; print $2 >"list2" }
```

writes the first field, **$1** , on the file **list1** , and the second field on file **list2** . The >> notation can also be used. For example:

```
print $1 >>"list"
```

appends the output to the file **list** . In each case, the output files are created if necessary. The filename can be a variable or a field as well as a constant. For example:

```
print $1 >$2
```

uses the contents of field 2 as a filename. There is a limit of ten possible output files. Output can also be piped into another process. For instance the following example mails the output to **fredm**'s mailbox:

```
print | "mail fredm"
```

The variables **OFS** and **ORS** may be used to change the current output field separator and output record separator. The output record separator is appended to the output of the print statement. **awk** also provides the **printf** statement for output formatting:

```
printf format, expr, expr, ...
```

formats the expressions in the list according to the specification in the file format and prints them. For example:

```
printf "%8.2f  %10ld\n", $1, $2
```

prints **$1** as a floating point number 8 digits wide, with two digits after the decimal point, and **$2** as a 10-digit decimal number, followed by a **NEWLINE** . No output separators are produced automatically; they must be added, as in the above example.

PATTERNS

You may specify a pattern before an action to act as a selector for determining whether the action is to be executed. A variety of expressions may be used as patterns: regular expressions, arithmetic relational expressions, string-valued expressions, and arbitrary Boolean combinations of these.

The special pattern **BEGIN** matches the beginning of the input, before the first record is read. The pattern **END** matches the end of the input, after the last record has been processed. **BEGIN** and **END** thus provide a way to gain control before and after processing, so you can initialize and terminate the program normally.

For example, the field separator can be set to a colon with:

```
BEGIN   { FS = ":" }
... rest of program ...
```

Or the input lines may be counted by:

```
END  { print NR }
```

If **BEGIN** is present, it must be the first pattern; **END** must be the last.

Regular Expressions

The simplest regular expression is a literal string of characters enclosed in slashes, such as:

```
/smith/
```

This is actually a complete awk program which prints all lines containing any occurrence of the name **smith** it will also be printed, as in:

```
blacksmithing
```

The list of regular expressions recognized by **awk** includes the regular expressions recognized by **ed** , **sed** , and the **grep** command. In addition, **awk** allows parentheses for grouping, the pipe (|) for alternatives, the plus (+) for "one or more", and the question mark (?) for zero or one. Character classes may be abbreviated: **[a-zA-Z0-9]** is the set of all letters and digits. For example, the **awk** program:

```
/[Aa]pples|[Bb]ananas|[Cc]herries/
```

prints all lines which contain any of the words apples, bananas, or cherries, whether they begin with an uppercase letter or not.

Regular expressions must be enclosed in slashes, just as in **ed** and **sed** . Within a regular expression, **BLANKS** and the regular expression special characters are significant. To turn off the special meaning of one of the regular expression special characters, precede it with a backslash.

For example, the pattern:

```
/\/.*\//
```

matches any string of characters enclosed in slashes. You can also specify that any field or variable matches a regular expression (or does not match it) with the operators tilde (~), and exclamation point tilde (!~). The program:

```
$1 ~ /[jJ]ohn/
```

prints all lines where the first field matches john or John. Notice that this will also match Johnson, St. Johnsbury, and so on. To restrict the match to exactly John or john, use the program:

```
$1 ~ /^[jJ]ohn$/
```

The caret (^) refers to the beginning of a line or field; the dollar sign ($) refers to the end.

Relational Expressions

An **awk** pattern can be a relational expression involving the operators <, <=, ==, !=, >=, and >. For example:

```
$2 > $1 + 100
```

selects lines where the second field is at least 100 greater than the first field. Similarly, the following example prints all lines with an even number of fields:

```
NF % 2 == 0
```

In relational tests, if neither operand is numeric, a string comparison is made; otherwise it is numeric. Thus:

```
$1 >= "s"
```

selects lines that begin with s, t, u, etc. In the absence of other information, fields are treated as strings, so the program:

```
$1 > $2
```

performs a string comparison.

Combinations of Patterns

A pattern can be any Boolean combination of patterns, using the operators || (or), && (and), and ! (not). For example:

```
$1 >= "s" .XX "& $1 < "t" && $1 != "smith"
```

selects lines where the first field begins with s, but is not smith. The operators && and || guarantee that their operands will be evaluated from left to right; evaluation stops as soon as their truth or falsehood is determined.

The pattern that selects an action may also consist of two patterns separated by a comma. In this example, the action is performed for each line between an occurrence of **pat1** and the next occurrence of **pat2** (inclusive):

```
pat1, pat2      { ... }
```

The next example prints all lines between **start** and **stop** :

```
/start/, /stop/
```

In the last example the action is performed for lines 100 through 200 of the input:

```
NR == 100,,NR == 200 { ... }
```

ACTIONS

In addition to the patterns described above, the **awk** program offers a set of possible actions. An **awk** action is a sequence of action statements terminated by **NEWLINES** or semicolons. These action statements can do a variety of bookkeeping and string manipulating tasks. The possible actions are: built-in functions, the assignment of variables and strings, the use of field variables, string concatenation statements, arrays, and flow-of-control statements.

Built-in Functions

awk provides a length function to compute the length of a string of characters. This program prints each record, preceded by its length:

```
{print length, $0}
```

The length by itself is a pseudo-variable which yields the length of the current record; **length(argument)** is a function which yields the length of its argument, as in the equivalent:

```
{print length($0), $0}
```

The argument may be any expression. **awk** also provides the arithmetic functions **sqrt** , **log** , **exp** , and **int** , for square root, logarithm,

exponential, and integer parts of their respective arguments. The name of one of these built-in functions, without argument or parentheses, stands for the value of the function on the whole record. The program:

```
length < 10 || length > 20
```

prints lines whose length is less than 10 or greater than 20.

The function **substr(s,m,n)** produces the substring of **s** that begins at position **m** (origin 1) and is at most **n** characters long. If **n** is omitted, the substring goes to the end of **s** . The function **index(s1, s2)** returns the position where the string s2 occurs in **s1** , or zero if it does not.

The function **sprintf(f, e1, e2, ...)** produces the value of the expressions **e1** , **e2** , etc., in the **printf** format specified by **f** . Thus, for example:

```
x = sprintf("%8.2f %10d", $1, $2)
```

sets **x** to the string produced by formatting the values of **$1** and **$2** .

Variables, Expressions, and Assignments

awk variables take on numeric (floating-point) or string values according to context. In the following example:

```
x = 1
```

x is clearly a number, while in the next example, **x** is clearly a string:

```
x = "smith"
```

Strings are converted to numbers and vice versa whenever context demands it. For instance:

```
x = "3" + "4"
```

assigns 7 to **x** . Strings which cannot be interpreted as numbers in a numerical context will generally have the numeric value zero.

By default, variables (other than built-in functions) are initialized to a null string, which has numerical value zero. This eliminates the need for most **BEGIN** sections. For example, the sums of the first two fields can be computed with:

```
      { s1 += $1; s2 += $2 }
END   { print s1, s2 }
```

Arithmetic is done internally in floating point. The arithmetic operators are: +, -, *, /, and %. The C increment ++ and decrement - - operators are also available, as well as the assignment operators +=, -=, *=, =, and %=. These operators may all be used in expressions.

Field Variables

Fields in **awk** share essentially all of the properties of variables. They may be used in arithmetic or string operations, and may be assigned to. Thus, you can replace the first field with a sequence number:

```
{ $1 = NR; print }
```

or accumulate two fields into a third:

```
{ $1 = $2 + $3; print $0 }
```

Or you can assign a string to a field as in the next example which replaces the third field by "too big" when it is too big, and prints the record in either case:

```
{ if ($3 > 1000)
        $3 = "too big"
  print
}
```

Field references may be numerical expressions, as in the following:

```
{ print $i, $(i+1), $(i+n) }
```

Whether a field is deemed numeric or string depends on context; in ambiguous cases like the following example, fields are treated as strings:

```
if ($1 == $2) ...
```

Each input line is automatically split into fields as necessary. It is also possible to split any variable or string into fields. For example:

```
n = split(s, array, sep)
```

splits the the string s into **array[1]** , **array[n]** . The number of elements found is returned. If the **sep** argument is provided, it is used as the field separator. Otherwise FS is used as the separator.

String Concatenation

Strings may be concatenated. For example:

```
length($1 $2 $3)
```

returns the length of the first three fields. The following **print** statement prints the two fields separated by "is":

```
print $1 " is " $2
```

Variables and numeric expressions may also appear in concatenations.

Arrays

Array elements are not declared; they spring into existence when mentioned in a program. Subscripts may have any non-null value, including non-numeric strings. For example, in a conventional numeric subscript, the statement:

```
x[NR] = $0
```

assigns the current input record to the NR th element of the array x . In principle it is possible to process the entire input in a random order with the **awk** program:

```
      { x[NR] = $0 }
END { ... program ... }
```

The first action merely records each input line in the array x .

Array elements may be named by non-numeric values. Suppose the input contains fields with values like apple and orange. The program:

```
/apple/          { x["apple"]++ }
/orange/         { x["orange"]++ }
END              { print x["apple"], x["orange"] }
```

increments counts for the named array elements, and prints them at the end of the input. Any expression can be used as a subscript in an array reference. Thus:

```
x[$1] = $2
```

uses the first field of a record as a string to index the array x .

Suppose each line of input contains two fields, a name and a nonzero value. Names may be repeated. To print a list of each unique name followed by the sum of all the values for that name, use the program:

```
      { amount[$1] += $2 }
END   { for (name in amount)
              print name, amount[name] }
```

To sort the output, replace the last line with the following:

```
print name, amount[name] | "sort"
```

Flow-of-Control Statements

Like any programming language, **awk** provides flow-of-control statements. These are: if-else, while, for, and statement groupings with braces. When using the if statement the condition in parentheses is evaluated. If it is true, the statement following the if is done. The else part is optional.

A while statement is also available. For example, to print all input

fields one per line, use:

```
i = 1
while (i <= NF) {
        print $i
        ++i
}
```

The for statement:

```
for (i = 1; i <= NF; i++)
        print $i
```

does the same job as the while statement above.

An alternate form of the for statement is useful for accessing the elements of an associative array. For example:

```
for (i in array)
        statement
```

performs statement with i set in turn to each element of the array. The elements are accessed in an apparently random order. Chaos will ensue if i is altered, or if any new elements are accessed during the loop.

The expression in the condition part of an if, while or for statement can include relational operators like <, <=, >, >=, == (is equal to), and != (not equal to; regular expression matches with the match operators \~ and !\~; the logical operators ||, &&, and !, and parentheses for grouping.

The break statement causes an immediate exit from an enclosing while or for statement. The continue statement causes the next iteration to begin. The next statement causes **awk** to skip immediately to the next record and begin scanning the patterns from the top. The exit statement causes the program to behave as if the end of the input had occurred.

One final note: comments may be placed in **awk** programs. If you are going to store complex **awk** programs for future use, it is a good idea to use comment lines generously, to remind you of what your program does:

```
print x, y      # this is a comment
```

Comments begin with the character # and end with the end of the line.

NOTICE

Ing. C. Olivetti & C., S.p.A. reserves the right to make any changes in the product described in this manual at any time and without notice.

This manual is licensed to the Customer under the conditions contained in the User License enclosed with the Program to which the manual refers.

Code 4022940 Y (0)
Printed in Italy

olivetti
PERSONAL
COMPUTER

olivetti