intel®

XENIX* 286 User's Guide

xenix

Order Number: 174387-003

XENIX* 286
USER'S GUIDE

Order Number: 174387-003

Above	iCEL	int$_e$l	iPDS	Megachassis	QUEST
BITBUS	iCS	int$_e$lBOS	iPSC	MICROMAINFRAME	QueX
COMMputer	iDBP	Intelevision	iRMX	MULTIBUS	Ripplemode
CREDIT	iDIS	int$_e$ligent Identifier	iSBC	MULTICHANNEL	RMX/80
Data Pipeline	iLBX	int$_e$ligent Programming	iSBX	MULTIMODULE	RUPI
Genius	i$_m$	Intellec	iSDM	ONCE	Seamless
i	iMDDX	Intellink	iSXM	OpenNET	SLD
i	iMMX	iOSP	Library Manager	Plug-A-Bubble	UPI
I^{2}ICE	Insite		MCS	PROMPT	VLSiCEL
ICE				Promware	

REV.	REVISION HISTORY	DATE
-001	Original issue	11/84
-002	Revision	08/85
-003	Revision	12/85

intel®

TABLE OF CONTENTS

CONTENTS

CONTENTS PAGE

CONTENTS **PAGE**

CONTENTS **PAGE**

CONTENTS **PAGE**

CONTENTS **PAGE**

intel®

CHAPTER 1 INTRODUCTION

Overview

This manual introduces the XENIX 286 Operating System and explains the fundamental concepts needed to use it effectively. Unless otherwise noted, this manual discusses the Basic System and the Bourne shell only.

The XENIX 286 system is an improved and enhanced version of the UNIX System III from Bell Laboratories. It is intended for use in schools, corporations, laboratories, and small office environments. XENIX is well known as a productive environment for software development and as a text processing environment.

Audience

Because XENIX 286 is designed to be used in a variety of environments, users of the system have a wide range of computer experience and education. Inexperienced users should read the *Overview of the XENIX 286 Operating System* first and progress to more advanced documentation. Experienced XENIX users will very likely be able to begin using the XENIX 286 system immediately, using the reference manual and user's guides as necessary.

Notation

These notational conventions are used in this manual:

- Literal names are bolded where they occur in text, e.g., **/sys/include, printf, dev_tab, EOF.**
- Syntactic categories are italicized where they occur and indicate that you must substitute an instance of the category, e.g., *filename*.
- In examples of dialogue with the XENIX 286 system, characters entered by the user are bolded.
- In syntax descriptions, optional items are enclosed in brackets (**[ ]**).
- Items that can be repeated one or more times are followed by an ellipsis (**...**).
- Items that can be repeated zero or more times are enclosed in brackets and followed by an ellipsis (**[ ]...**).
- A choice between items is indicated by separating the items with vertical bars (|).

The XENIX 286 Working Environment

An operating system efficiently organizes and controls the resources of a computer. These resources include memory, disks, line printers, terminals, and any other peripheral devices connected to the system. The heart of XENIX 286 is a multiuser, multitasking operating system. A multiuser system enables several users to use a computer simultaneously, thus providing lower cost in computing power per user. In a multitasking system, several programs can run simultaneously, thereby increasing productivity.

Because UNIX and XENIX have been accepted as standards for "high-end" operating systems, a great deal of software is available for this environment. In addition, XENIX 286 is a bridge to the MS-DOS operating system. For systems that support MS-DOS, XENIX 286 provides commands that enable you to access MS-DOS format files and disks. The XENIX 286 system also includes several widely praised enhancements developed at the University of California at Berkeley and a visual interface similar to other productivity tool interfaces.

Other characteristics of the XENIX 286 system include

- A powerful command language for programming XENIX 286 commands. Unlike other interactive command languages, the XENIX 286 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 286 includes commands that 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 desk calculator program.
- Mountable and dismountable file systems that facilitate addition of flexible disk drives to the system.
- A complete set of flexible directory and file protections that utilizes all combinations of read, write, and execute access for the owner of each file or directory, as well as for groups of users.
- Facilities for creating, accessing, moving, and processing files and directories in a simple and uniform way.

Using This Manual

This manual is organized as follows:

Chapter 1: Introduction
This chapter gives an introduction and overview of the XENIX 286 system and this manual.

Chapter 2: Tasks
This chapter explains how to perform basic tasks using appropriate XENIX 286 commands.

Chapter 3: The Shell
This chapter describes use of the shell command interpreter and how to write procedures that can be executed by the shell interpreter.

Chapter 4: **ed:** A Line-Oriented Text Editor
This chapter explains how to use the line editor, **ed.**

Chapter 5: **vi:** A Visual Text Editor
This chapter explains how to use the screen editor, **vi.**

Chapter 6: **mail:** The XENIX Mail System
This chapter describes the XENIX 286 mail facility and explains how to send and receive mail.

Chapter 7: **bc:** A Calculator
This chapter explains how to use **bc,** a sophisticated calculator program.

Appendix A: Related Publications
Lists Intel publications containing information related to the XENIX operating system.

This manual does not attempt to give information about installing, managing, and maintaining the system, nor does it discuss document preparation, software development, or any of the specialized utilities available in other XENIX 286 system products. Appendix A contains a list of manuals relating to these subjects.

intel®

CHAPTER 2 TASKS

Introduction

This chapter is designed to familiarize you with some basic XENIX commands and to show you how to perform such common tasks as logging in and out, manipulating files and directories, and processing data. The *XENIX 286 Reference Manual* contains a detailed entry for each of the commands discussed in this chapter. The *Overview of the Xenix 286 Operating System* contains a detailed description of the XENIX file system discussed in this chapter.

NOTE

This chapter makes reference to a DELETE key on your terminal. If your terminal does not have a DELETE key and you do not know which key on your terminal corresponds to a DELETE key, get help from the system administrator.

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. See your system administrator to get an account. This section assumes your account has already been set up.

Normally, the system sits idle and the prompt "login:" is displayed on the terminal screen. If your screen is blank or displays "garbage," press the RETURN key.

When the login prompt appears, follow these steps:

1. Type your login name, then press RETURN. If you make a mistake, press the BACKSPACE key to erase character by character. After you press RETURN, the word "Password:" appears on your screen.

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

If you have typed your login name and password correctly, a prompt or a menu appears on the screen. The prompt tells you that the XENIX system is ready to accept commands from the keyboard. When you log in, you are placed in an area called the login, or home, directory.

If you make a mistake while logging in, the system displays the message

```
Login incorrect
login:
```

If you get this message, log in again. You must type all the letters of your user name and password correctly before you can access the system.

Logging Out

The logout process depends on the system interface, called a shell in XENIX systems, that was activated when you logged in to the system. The standard shell is the Bourne shell. The XENIX system also has two other shells, the C shell and the Visual shell. You might be using one of these or a custom shell.

To log out of the system you must be in your login shell (activated when you logged in). If your login shell is a menu shell, there should be a menu selection or key to exit or quit. If your login shell is a prompting shell, type

```
exit
```

and press RETURN to log out. If another prompt appears after you enter **exit**, it means that the current shell was not your login shell. Continue entering **exit** until the prompt disappears and the login message is displayed.

Entering and Erasing a Command Line

Entering a command line consists of typing characters at a terminal, then pressing RETURN. When you press RETURN, the computer reads the command line and executes commands specified on that line. You may type as many command lines as you want without waiting for them to execute, because XENIX supports type-ahead of characters.

When entering commands, typing errors are bound to occur. To erase the errors character by character, use the BACKSPACE key.

Changing Your Password

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

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

1. Type

   ```
   passwd
   ```

   and press RETURN. The following message appears:

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

2. Carefully type your old password and press RETURN. Your password is not echoed on the screen. If you make a mistake, press RETURN. The message "Sorry" appears, then the system prompt. Begin again with step 1.

3. When you have typed your old password, this message appears:

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

   Type in your new password and press RETURN.

4. The message

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

   appears. Type your new password and press RETURN again. If you make a mistake, press RETURN. The message

   ```
   They don't match; try again.
   New password:
   ```

   appears, and you must begin again with step 3. When you have completed the procedure, the system prompt appears.

Manipulating Files

File manipulation (e.g., creating, displaying, combining, copying, moving, naming, and deleting files) is one of the most important capabilities an operating system provides. 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 one of the text editors described in Chapters 4 and 5. After you save the file and exit the text editor, use the **lc** command to see the file name listed in the directory.

Displaying File Contents

The **more** command displays the contents of a file one screen at a time and has the form

> **more** *options filename*

more is useful for looking at a file when you don't want to make changes to it. For example, to display the contents of the file **memos,** type

```
more memos
```

You can also invoke **more** with a number of options that control where the display begins and how the file is displayed. These options include

+linenumber	Begins the display at the line in the file designated by *linenumber*.
+/text	Begins the display two lines before *text*, where *text* is a word or number in the file. 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.**

For example, to begin looking at the file **memo** at the first occurrence of the words "net gain" type

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

If the file is more than one screen 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:

RETURN	Scrolls down one line.
d	Scrolls down one-half screen.
SPACE BAR	Scrolls down one full screen.
*n*SPACE BAR	Scrolls down *n* lines.
.	Repeats the previous command.
CONTROL-S	Stops screen output until another key is pressed.

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 RETURN. The message

```
...skipping
```

appears, XENIX displays the file from two lines before the line where "net gain" is, and the file scrolls up from the bottom of the screen.

If you use **more** to look at a file and decide you want to change the file, you can invoke the **vi** editor by typing

```
v
```

and pressing RETURN. The file must be large enough to require more than one screen to display it, otherwise **more** exits and the editor will not open the displayed file.

See Chapter 5, "vi: A Visual Text Editor," for 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 commands **head** and **tail** 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, typing

```
tail -4 memo
```

displays the last four lines of **memo.**

Like **more, cat** also displays the contents of a file, but **cat** scrolls to the end of the file unless you press CONTROL-S to stop it. Press another key to continue the scrolling. If you wish to abort the display before the end of the file, press the DELETE key. For example, to display the contents of **file1**, type

```
cat file1
```

To display the contents of **file1**, **file2**, and **file3**, type

```
cat file1 file2 file3
```

Combining Files

The **cat** command is frequently used to combine files into some other new file. The greater-than sign (>) is used to redirect the output of **cat** to the new file. Thus, to combine the two files named **file1** and **file2** in a new file named **bigfile,** type

```
cat file1 file2 >bigfile
```

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

```
cat file1 >> file2
```

Note that after appending **file1** to **file2, file1** still exists as a separate file.

Moving a File

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

```
mv text book
```

After this move is completed, no file named **text** exists in the working directory, because the file has been renamed **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 moved files are no longer in your working directory but are now in the directory **/tmp.**

The **mv** command always checks to see if the last argument is the name of a directory and, if so, all files designated are moved into that directory.

Renaming a File

To rename a file, simply move it to a file with the new name; the old name of the file is automatically removed. Thus, to rename the file **anon** to **johndoe,** type

```
mv  anon johndoe
```

Copying a File

The **cp** command is used for copying and has two forms: one to copy files into a directory and one to copy a file to another file. Thus, to copy three files into an existing directory named **filedir**, type

```
cp file1 file2 file3 filedir
```

The original versions of the three files still reside in the working directory, and the file names are identical in the two directories. Like **mv**, **cp** always checks to see if the last argument in the command line is the name of a directory and, if so, all designated files are copied into that directory.

To create two copies of a file in your working directory, you must rename the copy. To do this, **cp** can be invoked as follows:

```
cp file filecopy
```

After the above command has executed, 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 *filename*

Using the command

rm -i *filename*

causes XENIX to display the file name and wait for a yes or no response. To remove it, type

```
y
```

for "yes" and press RETURN. To leave it, type

```
n
```

for "no" and press RETURN. This command is useful when cleaning up a directory that contains many files.

Finding a File

The **find** command searches for a specified file and is useful for locating files with identical names or for finding a file when you don't know which directory it is in. The command has the form

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

where *pathname* is the path name of the directory you want to search and *filename* is the name of the file you are searching for. **find** searches recursively, that is, it starts at the named directory and searches downward through all files and subdirectories under the directory specified in *pathname*.

The -**name** option indicates that you are searching for a file with a specific file name. (Other search conditions used with **find** are described in the *XENIX 286 Reference Manual*.)

The -**print** option displays the path names of all files that match *filename*. By using the output redirection symbol (>), you can direct the output of **find** to a file rather than to the screen.

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 in different directories so that when a file is changed in one directory, it is also changed in the other directory. 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.

When you link files, a file name is associated with an inode. An inode is a number that specifies a unique set of data on the disk. One or more file names may be associated with this data. Thus, the above command assures that the files **/usr/joe/memos** and **/usr/mary/joememos** have identical contents.

Three rules to remember about linking files:

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

2. Linking files used by more than one person is risky, because anyone 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 four links, three links remain. For more information about linking files, see **ln** in the *XENIX 286 Reference Manual.*

Manipulating Directories

Because of the hierarchical organization of its file system, XENIX has many directories and subdirectories. The file system contains directories for each user. Within your user directory you can create, delete, and copy directories. Commands that facilitate directory manipulation are described in the following sections. The *Overview of the Xenix 286 Operating System* contains a detailed description of the XENIX file system.

Listing Directory Contents

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

lc *options directoryname*

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
```

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

-a Lists all files in the directory, including the "hidden" files (file names 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, the latest (most recently modified) first. When used with the **-r** option, lists the oldest first.

-R Lists all files and directories in the current directory, plus each file and directory *below* the current one. The "R" stands for "recursive."

-F Marks directories with a slash(/) and executable files with an asterisk (*).

-l Gives an expanded listing of a directory, producing an output that looks similar to the following:

```
total 501
drwxr-x---   2   boris   grp1    272   Apr 5 14:33   dir1
drwxr-x---   2   enid    grp1    272   Apr 5 14:33   dir2
drwxr-x---   2   iris    grp1    592   Apr 6 11:12   dir3
-rw-r-----   1   olaf    grp2    282   Apr 7 15:11   file1
-rw-r-----   1   olaf    grp2     72   Apr 7 13:50   file2
-rw-r-----   1   olaf    grp2   1403   Apr 1 13:22   file3
```

Reading from left to right, the information given for each file or directory includes

- Permissions
- Number of links
- Owner
- Group
- Size in bytes
- Date and time of last modification
- File or directory name

The information in this listing and how to change permissions are discussed in the section "Using File and Directory Permissions" later in this chapter.

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
```

After this command has been executed, a new empty directory will exist in your working directory.

Removing a Directory

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

```
rmdir phonenumbers
```

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

Renaming Directories

The **mv** command is used to rename directories. To rename the directory **little.dir** in your current directory to **big.dir,** type

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

This is a simple renaming operation; no files are moved. The directory **big.dir** must not already exist or XENIX will give you an error message.

Copying Directories

The **copy** command copies directories.

CAUTION

Do not attempt to copy the root, or /, directory because you are requesting that the entire file system be copied and it is unlikely that there is enough disk space for two copies of the entire file system.

This command has the form

copy *options olddirectory newdirectory*

To copy all the files in the directory **/usr/joe/memos** into **/usr/joe/notes,** enter

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

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 all the files and subdirectories under the named directory (recursively).

To copy all the files and subdirectories in the directory **/usr/joe/accts/30days** into **/usr/joe/accts/overdue,** enter

```
copy -r /usr/joe/accts/30days /usr/joe/accts/overdue
```

and the directory **30days** becomes the directory.

Moving in the File System

When using the XENIX system, it helps to imagine a large tree structure of files and directories. Each directory should be thought of as a place that you can move into or out of. At all times you are "some place" in the tree structure. This place is called your working or current directory. The commands used to find out where you are and to move around in the tree structure are discussed in the following sections. The *Overview of the Xenix 286 Operating System* contains a detailed description of the XENIX file system.

Where You Are

All commands are executed relative to the working directory. You can find out the name of this directory by using the **pwd** command, which stands for "print working directory." For instance, if your working directory is **/usr/joe,** when you type

```
pwd
```

you will get the output

```
/usr/joe
```

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

Changing Directories

To move to any other directory in the system, use the **cd** ("change directory") command and specify that directory as an argument to **cd.** For example, the command

```
cd /usr
```

moves you to the **/usr** directory.

To ascend the directory tree structure one level, type

```
cd ..
```

For example, if you are in the directory **/usr/joe/work** and issue the above command, you would move from **/usr/joe/work** to **/usr/joe.** Similarly, the command

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

moves you from **/usr/joe/work** to **/usr,** ascending two levels in the structure.

To return to your home directory from anywhere, type

```
cd
```

Using File and Directory Permissions

The XENIX system enables the owner to restrict access to files and directories, limiting who can read, write, and execute files owned by him or her. To determine the permissions associated with a given file or directory, use the l command. The output from the l command should look something like this:

```
total 501
drwxr-x---   2   boris   grp1    272   Apr 5 14:33   dir1
drwxr-x---   2   enid    grp1    272   Apr 5 14:33   dir2
drwxr-x---   2   iris    grp1    592   Apr 6 11:12   dir3
-rw-r-----   1   olaf    grp2    282   Apr 7 15:11   file1
-rw-r-----   1   olaf    grp2     72   Apr 7 13:50   file2
-rw-r-----   1   olaf    grp2   1403   Apr 1 13:22   file3
```

Permissions are indicated by the first ten characters of the output. The permissions for 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:

-	Indicates an ordinary file.
d	Indicates a directory.
c	Indicates a character special device such as a line printer or terminal.
b	Indicates a block special device such as a hard or flexible disk.
n	Indicates a name special file (i.e., a semaphore used for controlling access to some resource).
s	Indicates a shared data file.
p	Indicates a named pipe.

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

- Owner permissions
- Group permissions
- All other user permissions

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

Ordinary file permissions have the following meanings:

r	read permission
w	write permission
x	execute permission
-	no permission

For directories, permissions have the following meanings:

r	Files may be listed in the directory; the directory must have execute permission.
w	Files may be created or deleted in the directory; as with "r", the directory itself must also have execute permission.
x	The directory may be searched. A directory must have execute permission before you can move to it, access a file within it, or list the files in it.

The following are some typical directory permission combinations:

d---------	No access at all. This mode denies directory access to all users except **root** (a special account controlled by the system administrator.
drwx------	Allows access by the owner to use **lc**, create files, delete files, access files (subject to file permissions), and use **cd**. 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 use **cd** to move 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 use **cd** to change to the directory. Other users can only access a file within this directory by its exact name; they cannot search for a file by using metacharacters. Files cannot be created or deleted in the directory by anyone except the owner. This mode is rarely used, but it can be useful if you want to give someone access to a specific file in a directory without permitting access to other files in the same directory.

This chapter discusses ordinary files, executable files, and directories only. For information about other types of files, see **ls** in the *XENIX 286 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 three classes of users, and they are indicated as follows:

u	User, the owner of the file or directory
g	Group, the group the owner of the file belongs to
o	Other, all users of the system

All three classes of users may be designated by using the character "a" for "all".

For example, assume **file1** exists with the following permissions:

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

The owner of this file has read and write permission, group members have read permission, and all others have no access.
To give **file1** execute permission for all classes of users, type

```
chmod a+x file1
```

In the instruction segment of the command (a+x), the "a" stands for "all classes of users" and the "x" stands for execute permission. The resulting permissions are

```
-rwxr-x--x
```

To remove the owner's write and execute permissions and the group's execute permission on the above file, type

```
chmod ug-wx file1
```

Changing Directory Search Permissions

Directories also have an "x" (execute) permission. Since directories cannot be executed however, this attribute signifies search permission. If execute permission is denied to a user, then that user cannot even list the names of the files in the directory.

For example, the directory **dir1** has the following permissions:

```
drwxr-xr-x
```

To change permissions so that the group of "other" users can't examine **dir1,** type

```
chmod o-rx dir1
```

The new attributes for **dir1** are now

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

Processing Information

The following sections describe a number of XENIX utilities available for processing data.

Comparing Files

To compare two text files, use the **diff** command to print out those lines that differ between specified files. For example, suppose that a file named **men** has the contents

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

and that a file named **women** has the following contents:

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

The command

```
diff men women
```

produces the following results:

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

The 1c1 means that line 1 in the file **men** and line 1 in the file **women** must be changed to make the files the same. The second and fourth lines of the **diff** output are the lines that are different in the two files. The < indicates the line in the file **men** that is different from any line in the file **women.** The > indicates the line in the file **women** that is different from any line in the file **men.** The --- separates the lines from the file **men** from the lines from the file **women.**

Echoing Arguments

The **echo** command echoes arguments to the standard output. For example, typing

```
echo hello
```

produces

```
hello
```

on the screen. To output several lines of text, surround the echoed argument in double quotation marks and press RETURN 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 produces the output

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

Sorting a File

One of the most useful file processing commands is **sort.** By default, **sort** sorts the lines of a file according to the ASCII collating sequence (i.e., it alphabetizes them). For example, to sort a file named **phonelist,** type

```
sort phonelist
```

In the above case, the sorted contents of the file are displayed on the screen. To create a sorted version of **phonelist** named **phonesort,** type

```
sort phonelist >phonesort
```

Note that **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.**

Searching for a Pattern in a File

The **grep** (global search for regular expressions and print) command selects and extracts lines from a file, printing only those lines that match a given pattern. For example, to print out all lines in a file containing the word "tty38", type

```
grep 'tty38' filename
```

where *filename* is the name of the file that you want searched.

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

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

```
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 related forms **fgrep** and **egrep,** and the types of patterns (regular expressions) it can be used to search for, see **grep** in the *XENIX 286 Reference Manual.*

Counting Lines, Words, and Characters

wc ("word count") is a command for counting lines, words, and characters in a file; 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 option. 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.

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. Foreground processes are processes that you directly control from the keyboard. Processes 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. Background processes may be run at any time; you must be logged in to initiate a background process, but once the process has started running you may log out.

Occasionally, you may need to know who is on the system or what processes are running before you can perform a task; this section includes procedures to determine this information.

Determining 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
```

The **who** command produces output similar to the following:

```
arnold      tty02      Apr 7 10:02
daphne      tty21      Apr 7 07:47
elliot      tty23      Apr 7 14:21
ellen       tty25      Apr 7 08:36
gus         tty26      Apr 7 09:55
adrian      tty28      Apr 7 14:21
```

Determining What Processes Are Running

Because commands can be placed in the background for processing, it is not always obvious which processes you are responsible for. The **ps** command stands for "process status" and displays information about currently running processes associated with your terminal. For instance, the output from a **ps** command might look like this:

```
PID     TTY     TIME     CMD
3459    c3      0:15     -sh
4831    c3      1:52     cc  program.s
5185    c3      0:00     ps
```

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

To find out all the processes running on the system, use the **-e** option:

```
ps -e
```

To find out about the processes running on another terminal, use the **-t** option and specify the terminal. For example, to find out what processes are associated with terminal c3, type

```
ps -tc3
```

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. These are called foreground processes. A background process, in contrast, need not finish executing before you give the next command. Background commands are 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&
```

The number of the process is displayed on the screen and 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 worked.

When processes are placed in the background, you have no control of them as they execute. For instance, pressing the DELETE key does *not* abort a background process. Instead, you must use the **kill** command described in the following section.

Killing a Process

To stop execution of a foreground process, press the DELETE key. This kills whatever foreground command is currently running. By using the **ps** command, you can determine the PID number of all foreground and background processes that you have running and then selectively kill any processe that you by using the the **kill** command and the process identification number (PID). To use the **kill** command in this way, first invoke the **ps** command and determine PID numbers. Select the processes you wish to kill, note the PID number, and issue the **kill** command by using the following format:

```
kill PID
```

If a subsequent **ps** shows that the process is still alive, use the -9 option in the following format for a sure kill:

```
kill -9 PID
```

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

Using the Line Printer

The following sections describe the commands to help you use a line printer effectively and efficiently.

Sending a File to the Line Printer

One of the most common operations that you will want to perform is printing files on the line printer. The most straightforward method for doing this is to type

```
lpr filename
```

for a single file, or

```
lpr filename1 filename2 filename3
```

for multiple files. Other common uses of **lpr** involve pipes. For example, to paginate and print a file of raw text, type

```
pr textfile | lpr
```

The **pr** (print to screen) and **lpr** (print to line printer) commands are very often used together. As another example, to sort, paginate, and print a file, type

```
sort datafile | pr | lpr
```

Getting Line Printer Information

At times it may be necessary to know the status of your print requests. You can view this information by using the **lpq** command. Type

```
lpq
```

and press RETURN.

Communicating with Other Users

Because XENIX supports multiple users, communicating with other users is easy and convenient. The various communications facilities are described in the following sections.

Sending mail

The XENIX **mail** program is a systemwide facility that enables system users to send and receive mail. To send mail to another user on the system, type

```
mail username
```

where *username* is the name of any system user. You may be asked to enter a subject for the message. If so type a brief (less than one line) subject and press RETURN. Enter the text of the message you want to send. Terminate text entry and send the message by typing a CONTROL-D on a blank line at the end of the message.

A complete **mail** session might go like this:

```
mail joe
There will be a meeting at 2:00 today to review recent developments with the new system.
CONTROL-D
```

Note that your XENIX system might ask for a subject before you enter the message.

For practice, send mail to yourself. (This isn't as strange as it might sound--mail to yourself is a handy reminder mechanism.) You can also send a previously prepared letter, and you can send mail to a number of people all at once. For more details, see Chapter 6, "mail: The XENIX Mail System," and the **mail** entry in the *XENIX 286 Reference Manual.*

Receiving mail

When you log in, you may sometimes get the message

```
you have mail
```

To read your mail, type

```
mail
```

A heading for each message is then displayed. To read the messages, press RETURN. The system displays one message at a time; the most recent message is displayed first. After reading each message, press RETURN again to read the next message.

After each message is displayed, **mail** waits for you to tell it what to do with the message. The two basic responses are **d,** which deletes the message, and RETURN, which stores the message in your **mbox** file or in your system mailbox, depending on how you exit **mail.** To exit mail, type **q** for "quit", **exit**, or CONTROL-D. Other responses are described in Chapter 6 of this manual and in the *XENIX 286 Reference Manual* under **mail.**

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
```

and press RETURN. If you get the reponse "permission denied", it means that joe has used the **mesg** command to deny other users access to his terminal. Otherwise, after you have executed the command by pressing RETURN, each subsequent line that you type is displayed both on your screen and on Joe's. When the message appears on Joe's screen, it is mixed with any other text or files currently being displayed, but the message does not affect the file itself. To terminate writing to Joe, enter a CONTROL-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

To display the time and date, type

```
date
```

Displaying a Calendar

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

cal *month year*

For example, to display a calendar for March 1952, type

```
cal 3 1952
```

The month may be expressed as a digit or as a month name. If you decide to use the specific name rather than a digit, you may abbreviate the month, using standard three-letter abbreviations. To display the calendar for an entire year, leave out the month. The year must 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 available for all XENIX system users. You can use the service by creating a file named **calendar** in your home or login directory. Each line in the file should have the following form:

```
date  text
```

Where *date* must be some form of month followed by day (e.g. Sep 7, Sept. 7, September 7, 9/7) and *text* can be any combination of characters. A typical **calendar** file might look like this:

```
8/16  Status Reports Due Today
9/20  Review meeting at 2:00 in conference room 200
9/1  Karen's  birthday
10/3  License  renewal
8/22  Pack camping gear for this weekend
9/16  Trip and expense reports are due
```

Each day your **calendar** file is examined and all of the lines whose dates match the current system date are placed in a message and mailed to you.

If you want to display the lines in your **calendar** file whose dates match the current system date, type

```
calendar
```

and press RETURN.

Calculating

The **bc** command invokes an interactive desk calculator that can be used as if it were a hand-held calculator. A practice session with **bc** is shown below. While **bc** does allow you to enter comments in the form **/*** text ***/**, it is not necessary to type them in if you are going to try the operations shown below. If you make a mistake typing something while in **bc**, use the BACKSPACE key to erase character by character.

```
/* This is a comment. bc will not attempt to process anything enclosed like this comment. */
/* Be sure to enter "scale = 0", otherwise the results will differ from those shown here. */

scale = 0
123.456789 + 987.654321 /* Add and output */
1111.111110
9.0000000 - 9.0000001 /* Subtract and output */
-.0000001
64/8 /* Divide and output */
8
1.12345678934 * 2.3    /* Multiply and output; note precision */
2.58395061548
19%4 /* Find remainder */
3
3^4   /* Exponentiation */
81
2/1*2 /* Note precedence  */
4
2/(1*2) /* Note precedence again */
1
x = 46.5   /* Assign value to x */
y = 52.5 /* Assign value to y  */
x + y + 1.0000 /* Add and output */
100.0000
obase = 16 /* Set hex output base */
15 /* Convert to hex */
F
16  /* Convert to hex */
10
64 /* Convert to hex */
40
255 /* Convert to hex */
FF
256 /* Convert to hex */
100
512 /* Convert to hex */
200
quit /* Must type whole word */
```

For more information, see Chapter 7, "**bc:** A Calculator."

intel®

CHAPTER 3 THE SHELL

Introduction

When first logging into XENIX, you communicate with the shell command interpreter, **sh.** This interpreter is a XENIX program that supports a very powerful command language. Each invocation of this interpreter is called a shell, and each shell has one function: to read and execute commands from the user.

Because the shell provides users with a high-level language to communicate with the operating system, XENIX can perform complex tasks not possible with less sophisticated operating systems. Commands that would normally 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 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 another command. This provides flexibility in tailoring a task for a particular purpose.

Basic Concepts

The shell itself (that is, the program that reads your commands when you log in or that is invoked with the **sh** command) is a program written in the C language; it is not part of the operating system proper, but an ordinary user program.

The Shell

In XENIX, a process is an executing task 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 "parent" processes.

Users log in to 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 XENIX, files may be created in one phase and then processed in the "background," enabling the user to continue working while programs run.

Commands

The most common way of using the shell is by typing simple commands at the keyboard. A "simple command" is any sequence of arguments separated by spaces or tabs. The first argument (numbered zero) specifies the name of the command to be executed. Any remaining arguments, with a few exceptions, are passed as arguments to that command. For example, the following command line might be typed to print the files **allan, barry,** and **calvin:**

```
$ lpr allan barry calvin
```

The dollar sign ($) is the standard Bourne shell prompt. The Bourne shell presents you with its prompt when it is waiting for input. Do not type the $. Note that in this chapter, user input (what you type) is shown in **bold.** Output from the computer is in regular type.

If the first argument of a command (in the above example, **lpr**) names an executable file (as indicated by an appropriate set of permission bits associated with that file) and is actually a compiled program, the shell as parent creates a child process that immediately executes that program. If the file is marked as being executable but is not a compiled program, it is assumed to be a shell procedure, that is, a file of ordinary text containing shell command lines. In this case, the shell spawns another instance of itself (a subshell) to read the file and execute the commands inside it.

From the user's viewpoint, compiled programs and shell procedures are invoked in exactly the same way. The shell determines which implementation has been used, rather than requiring the user to do so. This provides uniformity of invocation.

How the Shell Finds Commands

The shell normally searches for commands in three distinct locations in the file system: *command_name*, */bin/command_name*, and */usr/bin/command_name*. First, the shell attempts to use the command name as given; if this fails, it prepends the string **/bin** to the command name; and if this fails, it prepends **/usr/bin** to the command name. The effect is to search, in order, the current directory, then the directory **/bin,** and finally, the directory **/usr/bin.**

For example, the **pr** command is actually the file **/bin/pr.** A more complex path name may be given, either to locate a file relative to the user's current directory or to access a command with an absolute path name. If a given command name begins with a slash (for example, **/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. The particular sequence of directories searched may be changed by resetting the shell PATH variable. (Shell variables are discussed later in this chapter.)

Generating Argument Lists

The arguments to commands are very often file names. Sometimes, these file names are similar, but not identical. To take advantage of this similarity in names, the shell enables the user to specify patterns that match the file names in a directory. If a pattern is matched by one or more file names in a directory, then those file names are automatically generated by the shell as arguments to the command.

Most characters in such a pattern match themselves, but there are also XENIX special characters (metacharacters) that may be included in a pattern. These metacharacters are the following.

*	Matches any string regardless of length or content.
?	Matches any single character.
[ ]	Matches any of the enclosed characters or range of characters.

Here are some examples of metacharacter usage.

*	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**

This pattern-matching capability saves typing and, more importantly, makes it possible to organize information in large collections of files named in a structured fashion, using common characters or extensions to identify related files.

Pattern-matching has some restrictions. If the first character of a file name is a period (.), it can be matched only by an argument that literally begins with a period. If a pattern does not match any file names, then the pattern itself is printed out as the result of the match.

Note that directory names should not contain any of the following characters: * ? [ ]. If these characters are used, then infinite recursion may occur during pattern matching attempts.

Quoting Mechanisms

The characters <, >, *, ?, [, and] have special meanings to the shell. Removing the special meaning of these characters requires some form of quoting. This is done by using single quotation marks (') or double quotation marks (") to surround a string. A backslash (\) before a single character also provides this function.

All characters within single quotation marks are taken literally. Thus,

```
$ echostuff = 'echo $? $*; ls * | wc'
$ echo $echostuff
echo $?  $*; ls * | wc
```

The specified string is assigned to the variable **echostuff**, but it does *not* result in any other commands being executed.

Within double quotation marks, the special meaning of certain characters does persist, while all other characters are taken literally. The characters that retain their special meaning are the dollar sign ($), the backslash (\), the single 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 RETURN causes that RETURN to be ignored and is equivalent to a space. The backslash-RETURN pair is therefore useful in allowing continuation of long command lines.

Redirecting Input and Output

In general, most commands cannot determine 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).

Standard Input and Output

When a command begins execution, it usually expects that three files are already open: a "standard input", a "standard output", and a "diagnostic output" (also called "standard error"). A number called a *file descriptor* is associated with each of these files. By convention, file descriptor 0 is associated with the standard input, file descriptor 1 with the standard output, and file descriptor 2 with the diagnostic output. A child process normally inherits these files from its parent; all three files are initially connected to the terminal (0 to the keyboard, 1 and 2 to the screen). The shell enables the files to be redirected elsewhere before control is passed to an invoked command.

An argument to the shell of the form **<file** or **>file** opens the specified file as the standard input or output (in the case of output, destroying the previous contents of **file**, if any). An argument of the form **>>file** directs the standard output to the end of **file**, thus providing a way to append data to the file without destroying its existing contents. In either case, the shell creates **file** if it did 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 file names 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 <?
```

Note that special characters are *not* expanded in redirection arguments, because redirection arguments are scanned by the shell *before* pattern recognition and expansion takes place.

Diagnostic and Other Outputs

Diagnostic output from XENIX commands is normally directed to the file associated with file descriptor 2. (You may often need an error output file different from standard output so that error messages are not lost down pipelines.) You can redirect this error output to a file by immediately prepending the number of the file descriptor (2 in this case) to either output redirection symbol (> or >>). The following line appends error messages from the **cc** command to the file named **ERRORS.**

```
        $cc testfile.c 2>>ERRORS
```

Note that the file descriptor number must be prepended to the redirection symbol *without* any intervening spaces or tabs; otherwise, the number will be passed as an argument to the command.

This method may be generalized to allow redirection of output associated with any of the first ten file descriptors (numbered 0-9). For instance, if **cmd** puts output on file descriptor 9, then the following line will direct that output to the file **savedata.** (**cmd** is used in a generic sense here; there is no shell command called **cmd.**)

```
        $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 and 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 can execute in parallel, each program needs to read the output of its predecessor. For example, many commands operate on individual lines of text, reading a line, processing it, writing it out, and looping back for more input. Some must read large amounts of data before producing output; **sort** is an example of the extreme case that requires all input to be read before any output is produced.

The following is an example of a typical pipeline.

```
$ nroff -mm text | col | lpr
```

nroff is a text formatter available in the XENIX 286 Extended System that allows reverse line motions within its output; **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 **nroff** program provides a set of primitive commands as well as the capability of writing macros (sequences of primitive commands that can be called as a unit). The XENIX operating system provides a library of such macros. The flag **-mm** tells **nroff** to use the **mm** macro package. Another macro package is **ms**, and you could invoke **nroff** with this macro package by issuing the command,

```
$ nroff -ms text | col | lpr
```

The following examples illustrate some effects obtainable by combining commands.

who	Prints the list of logged-in users on the screen.
who >>log	Appends the list of logged-in users to the end of file **log**.
who | wc -l	Prints the number of logged-in users.
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 user names contain the string "bob". Note that the string consists of the following three letters: bob. The string does not include the double quotes. The convention followed by most XENIX documentation is to set off a string with double quotes.

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 or the XENIX operating system could misinterpret the command line.

who | sed '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.)

sed is the XENIX stream editor. It reads files and performs certain specified editor commands on each line of those files. In this example, the standard output of the command **who** is piped into **sed. sed** then runs the editing script **'s/ .*//'** on that output. This script saves only the first word of each line of **who**'s output, so that the input to **sort** is a file consisting of user names, one to a line. The program **sort** then arranges those lines in ASCII order and provides its output as input to the program **uniq.** With the **-d** option, this program saves only those lines that are repeated. The result is that you have a list of only those users that are logged in more than once.

The **who** command does not *by itself* provide options to yield all these results--they are obtained by combining **who** with other commands. Note that **who** just serves as the data source in these examples. As an exercise, replace **who |** with **</etc/passwd** in the above examples to see how a file can be used as a data source in the same way. Notice that redirection arguments may appear anywhere on the command line, even at the start. This means that

```
$ <infile >outfile sort|pr
```

is the same as

```
$ sort |pr <infile >outfile
```

Command Substitution

Any command can be placed within back quotation marks (sometimes called grave accents, not to be confused with single quotes) so that the output of the command replaces the backquoted command line itself. This concept is known as *command substitution.* The command or commands enclosed within back quotation marks are first executed by the shell and then their output replaces the whole expression, back quotation marks and all. This feature is often used to assign shell variables. (Shell variables are described in the next section.) For example, to assign the output of the **date** command to the shell variable **today,** type

```
$ today = `date`
```

The result is that the shell variable **today** has a value such as "Tue Nov 27 16:01:09 EST 1982". To display **today,** use the **echo** command.

```
$ echo $today
Tue Aug 14 16:01:09 PDT 1985
```

Any command that writes to the standard output can be enclosed in back quotation marks. Back quotation marks may be nested, but the inside sets must be escaped with backslashes. For example,

```
$ logmsg = `echo  Your  login  directory  is \`pwd\``
$ echo $logmesg
Your login directory is /usr/vrs
```

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 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 is in position zero on the command line and is assigned to the positional parameter **$0.** The first command argument is called **$1,** and so on. Within a shell script, the **shift** command may be used to access arguments in positions numbered higher than nine. For example, the following shell script might be used to cycle through command line switches and then process all succeeding files.

If you are unclear about how to enter and run a shell script, read the following indented text.

> Before runing this example, here are some basics on how to create an executable shell script. To create a shell script, invoke a text editor such as **vi.** Call the file you are creating whatever you want. This example assumes the name **myfile.** Enter the command,
>
> ```
> $ vi myfile
> ```
>
> The screen clears and tildes appear on the left. Press the i key for insert. The i does not appear on the screen, but you are now in insert mode. Type the file, just as you would on a typewriter. See Chapter 5 for a discussion of **vi** editing commands. Use those editing commands to correct any typing errors you might make. Press the ESC key to exit insert mode and reenter command mode. Then, type a colon; a colon appears on the last line of your screen. Type x for exit and follow it with a RETURN.
>
> Your script now exists in the file called **myfile.** To make it executable, enter the command,
>
> ```
> $ chmod u+x myfile
> ```
>
> The command **chmod** stands for change mode. The **u** stands for user (that's you), and the **+x** means that you are adding execution mode. Do not type spaces between the three characters, **u+x.**

The example shell script looks as follows.

```
while test $# != 0
do
        case $1 in
        -a) echo -a ; shift ;;
        -b) echo -b ; shift ;;
        -c) echo -c ; shift ;;
        -*) echo bad option ; exit 1 ;;
        *) echo process the rest of the command; shift ;;
        esac
done
```

Here's what the shell script does. First, it tests whether the number of arguments is zero. The number of arguments is represented by **$#**. The symbol **!=** means "not equal." While the number of arguments is not zero, the statements within the **do-done** delimiters are executed.

In this example, there is only one statement within the **do-done** delimiters, a **case** statement. The **case** statement begins with the word **case** and ends with the word **esac** (case spelled backwards). If the first argument (represented as **$1**) is **-a**, then the shell script writes **-a** to the screen. (In a more realistic example, you would probably want to do something more involved with that option.) The **shift** statement shifts the numbering of the arguments. **$1** goes away and **$2** becomes **$1, $3** becomes **$2,** etc. As long as there are arguments, the case statement is executed.

The * stands for any character or characters, including no characters. For example, if the argument were **-d**, the shell script would write "bad option" to the screen and exit. Options are identified by the minus sign. If the argument were **def**, the shell script would write "process the rest of the command" to the screen and shift. When the script shifts beyond the last argument, **$#** becomes zero, and the script terminates. Here's how the screen looks if you run this script in a test case.

```
$ myfile -a -b def
-a
-b
process the rest of the command
$
```

One can explicitly force values into the positional parameters by using the **set** command. For example,

```
$ set abc def ghi
$ echo $*
abc def ghi
```

assigns the string "abc" to **$1** (the first positional parameter), 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. The **echo** command displays those $ variables; **$*** means display all of them.

Using the **set** command requires some background understanding. For example, if you issue the command,

```
$ set -a -b def
-a: bad option(s)
```

you get an error message. That's because the **set** command sees the minus sign in front of the **a** and tries to interpret the **a** as one of its own options. The **set** command has no such option, and an error message results. Read the set section under the **sh** entry in the *XENIX 286 Reference Manual* to find out about valid **set** options. To prevent **set** from interpreting the **-a** as an option, add another minus sign, separated by spaces.

```
$ set - -a -b def
$ echo $*
-a -b def
```

Now you might think that if you invoke **myfile** without any options, you should get the same result as before. You set the dollar variables with **set** rather than providing them as arguments to **myfile.** The result should be the same. That's not what happens, though. If you execute **myfile,** the Bourne prompt returns with no action. **$#** was 0; your **myfile** did not see any positional variables. The reason why is key to understanding how shell scripts work.

When you execute **myfile,** it runs as a shell underneath the present shell. Those dollar variables set with the **set** command are not exported to **myfiles**'s shell. Try the following.

```
$ set - -a -b def
$ echo $*
-a -b def
$ sh
$ echo $*

$ exit
$ echo $*
-a -b def
```

The **sh** command spawns a new Bourne shell. The positional variables are not defined in this spawned shell. Hence, you see nothing when you execute **echo $*.** You can return to the spawning shell with the **exit** command. (A CONTROL-D works just as well.) Note that your positional parameters are still defined.

When you execute a shell script, it runs in a spawned shell. It does not get the positional parameters you set with a **set** command executed in the spawning shell.

User-Defined Variables

The shell also recognizes alphanumeric variables to which string values may be assigned. A simple assignment has the syntax,

name=string

Thereafter, **$name** will yield the value *string.* A *name* is a sequence of letters, digits, or underscores that begins with a letter or an underscore. No spaces surround the equal sign (=) in an assignment statement. Positional parameters may not appear on the left side of an assignment and can only be set as described in the previous section.

More than one assignment may appear in an assignment statement, but note that *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 RETURNs to be included in a string, while also allowing variable substitution (also known as "parameter substitution") to occur. This means that references to positional parameters and other variable names 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 previous example, the variable **echovar** has as its value the string consisting of the values of the first four positional parameters, separated by spaces, plus the string "echo". No quotation marks are needed around the string of asterisks being assigned to **stars** because pattern matching (expansion of star, 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, so that the following example results in **$first** and **$second** having the same value.

```
$ first = 'a  string  with  embedded  spaces'
$ second = $first
```

In accessing the values of variables, you may enclose the variable name in braces {...} to delimit the variable name from any following string. In particular, if the character immediately following the name is a letter, digit, or underscore, then the braces are required. For example, examine the following input.

```
$ a = 'This  is  a  string'
$ echo  "${a}ent  test  of  variables."
```

Here, the **echo** command prints

```
This is a stringent test of variables.
```

If no braces were used, the shell would substitute a null value for **$aent** and print

```
test of variables.
```

The following variables are maintained by the shell. Some of them are set by the shell, and all of them can be reset by the user.

HOME Initialized by the **login** program to the name of the user's login directory, that is, the directory that becomes the current directory upon completion of a login; **cd** without arguments switches to the $HOME directory. Using this variable helps keep full path names out of shell procedures. This is beneficial when path names are changed, either to balance disk loads or to reflect administrative changes.

IFS The variable that specifies which characters are internal field separators. These are the characters the shell uses during blank interpretation. (If you want to parse some delimiter-separated data easily, you can set IFS to include that delimiter.) The shell initially sets IFS to include the blank, tab, and RETURN characters.

MAIL The path name 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 (for example, 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 The variable that specifies the search path used by the shell when looking for commands. Its value is an ordered list of directory path names 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 (for example, **$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 The variable that specifies what string is to be used as the primary prompt string. If the shell is interactive, it prompts with the value of PS1 when it expects input. The default value of PS1 is "$ " (a dollar sign followed by a blank).

PS2 The variable that specifies the secondary prompt string. If the shell expects more input when it encounters a RETURN 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 spawned from your login file. Use **export** at the end of your **.profile** file. An example of an **export** statement follows.

```
export HOME IFS MAIL PATH PS1 PS2
```

Remember that, unless you export them, shell variables are not recognized in spawned shells. The **export** statement ensures that the specified variables are recognized in all spawned shells. XENIX programmers would say that the variables are "exported" to the spawned shells. Use the **set** command to view variables in the current shell. Use the **env** command to view variables that are exported.

Predefined Special Variables

Several variables have special meanings and are set *only* by the shell.

$# Records the number of arguments passed to the shell, not counting the name of the shell procedure itself. For instance, **$#** yields the number of the highest set positional parameter. Thus

```
sh cmd a b c
```

automatically sets **$#** to 3. One of its primary uses is in checking for the presence of the required number of arguments.

```
if test $# -lt 2
then
        echo 'two or more args required'; exit
fi
```

$? Contains the exit status of the last command executed (also referred to as "return code", "exit code", or "value"). Its value is a decimal string. Most XENIX commands return zero to indicate successful completion. The shell itself returns the current value of **$?** as its exit status.

$$ The process number of the current process. Because process numbers are unique among all existing processes, this string is often used to generate unique names for temporary files. XENIX provides no mechanism for the automatic creation and deletion of temporary files; a file exists until explicitly removed. Temporary files are generally undesirable objects; the XENIX pipe mechanism is far superior for many applications. However, the need for uniquely-named temporary files does occasionally occur.

The following example illustrates the recommended practice of creating temporary files; note that the directories **/tmp** and **/usr/tmp** are cleared out if the system is rebooted. A # indicates that what follows on that line is a comment.

```
#     use current process id
#     to form unique temp file
temp=/usr/tmp/$$
ls >$temp
#     commands here, some of which use $temp
rm $temp
#     clean up at end
```

$! The process number of the last process run in the background (using the ampersand (&)). This is a string containing from one to five digits.

$- A string consisting of names of execution flags currently turned on in the shell. For example, **$-** might have the value **xv** if you are tracing your output.

The Shell State

The state of a given instance of the shell includes the values of positional parameters, user-defined variables, environment variables, modes of execution, and the current working directory.

The state of a shell may be altered in various ways. These include changing the working directory with the **cd** command, setting several flags, and reading commands from the special file, **.profile**, in your login directory.

Changing Directories

The **cd** command changes the current directory to the one specified as its argument. This can and should be used to change to a convenient place in the directory structure. Note that **cd** is often placed within parentheses to cause a subshell to change to a different directory and execute some commands without affecting the original shell.

For example, the first command below copies the file **/etc/passwd** to **/usr/***username***/passwd**; the second command changes directory to **/etc** and then copies the file. *username* represents your login directory. For the sake of this example, assume that your username (hence your login directory) is called **vrs.** People often use their initials--first, middle, and last.

```
$ cp /etc/passwd /usr/vrs/passwd
$ (cd /etc ; cp passwd /usr/vrs/passwd)
```

Note the use of parentheses. Both commands have the same effect. Note, however, that the second line is executed as a separate shell. When it is done, you are still in your original directory.

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 keyboard.

Execution Flags

The **set** command enables you to alter the behavior of the shell by setting certain shell flags. In particular, the -**x** and -**v** flags may be useful when invoking the shell as a command from the terminal. The flags -**x** and -**v** may be **set** by typing

```
set -xv
```

The same flags may be turned off by typing

```
set +xv
```

These two flags have the following meanings.

-v Input lines are printed as they are read by the shell. This flag is particularly useful for isolating syntax errors. The commands on each input line are executed after that input line is printed.

-x Commands and their arguments are printed as they are executed. (Shell control commands, such as **for** and **while**, are not printed, however.) Note that -**x** causes a trace of only those commands 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.

Command Environment

All variables and their associated values 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 only placed in the environment of the procedure being invoked.

For example, consider the following shell script. The shell variables **a** and **b** are defined on the same line where **keycommand** is invoked. The variables are defined in **keycommand** 's environment, not the shell's.

```
# keycommand,
echo $a $b
$ a = key1  b = key2  keycommand
key1  key2
```

If you issue the **set** command, you will not see **a** and **b** listed as a shell variable, and you cannot display them with the **echo** command. 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 get a list of variables that have been made read-only. 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. Use the following syntax, depending on your application:

sh *process* [*argument...*]
A new instance of the shell is explicitly invoked to read *process*. Arguments, if any, can be manipulated.

sh -v *process* [*argument...*]
This is equivalent to putting **set -v** at the beginning of *process*. It can be used in the same way for the **-x, -e, -u,** and **-n** flags. Refer to the **sh** entry in the *XENIX 286 Reference Manual* for an explanation of these flags.

process [*argument...*]
If *process* is an executable file and is not a compiled executable program, the effect is similar to that of

sh *process argument*

An advantage of this form is that variables that have been exported in the shell will still be exported from *process* when this form is used (because the shell only forks to read commands from *process*). Thus any changes made within *process* to the values of exported variables will be passed on to subsequent commands invoked from *process*.

Passing Arguments to Shell Procedures

When a command line is scanned, any character sequence of the form $*n* is replaced by the *n*th argument to the shell, counting the name of the shell procedure itself as **$0.** This notation permits direct reference to the procedure name and to as many as nine positional parameters. Additional arguments can be processed using the **shift** command or by using a **for** loop.

The **shift** command shifts positional parameters to the left; that is, the value of **$1** is thrown away, **$2** replaces **$1, $3** replaces **$2,** and so on. The highest-numbered positional parameter becomes **unset** (**$0** is never shifted). For example, in the shell procedure **ripple** below, **echo** writes its arguments to the standard output. Lines that begin with a number sign (#) are comments.

```
# ripple command
while test $# != 0
     do echo $1 $2 $3 $4 $5 $6 $7 $8 $9
     shift
done
```

If the procedure were invoked with

```
ripple a b c
```

it would print

```
a b c
b c
c
```

The special shell variable "star" (**$***) causes substitution of all positional parameters except **$0.** Thus, the **echo** line in the **ripple** example above could be written more compactly as

```
echo $*
```

These two **echo** commands are *not* equivalent: the first prints at most nine positional parameters; the second prints *all* of the current positional parameters. The shell star variable (**$***) is more concise and less error-prone. One obvious application is in passing an arbitrary number of arguments to a command from within a shell script. For example,

```
wc $*
```

counts the words of each of the files named on the command line.

It is important to understand the sequence of actions used by the shell in scanning command lines and substituting arguments. The shell first reads input up to a RETURN or semicolon, and then parses that much of the input. Variables are replaced by their values and then command substitution (via back quotation marks) is attempted. I/O redirection arguments are detected, acted upon, and deleted from the command line. Next, the shell scans the resulting command line for internal field separators, that is, for any characters specified by IFS to break the command line into distinct arguments; explicit null arguments (specified by "" or '') are retained, while implicit null arguments resulting from evaluation of variables that are null or not set are removed. Then, file name generation occurs with all metacharacters being expanded. The resulting command line is then executed by the shell.

Sometimes command lines are built inside a shell procedure. In this case, it is useful to have the shell rescan the command line after all the initial substitutions and expansions have been performed. The **eval** command is available for this purpose. **eval** takes a command line as its argument and simply rescans the line, performing any variable or command substitutions specified. For example,

```
$command=who  output=' | wc -l'
$eval  $command  $output
```

results in the execution of the command line,

```
$who | wc -l
```

The first word of a line is always evaluated. You want the shell to rescan the line to evaluate the variable **$output** as well. If you leave out the **eval**, just the **who** will be executed. That's because **who** accepts an unevaluated **$output.** If you enter **who** followed by any text string, you get login information about yourself. Try it.

```
$who junk morejunk
vrs        ttyc2    Aug 15  10:25
```

The output of **eval** cannot be redirected. However, uses of **eval** can be nested, so that a command line can be evaluated several times.

Directing the Flow of Control

The shell provides several commands that implement a variety of control structures useful in controlling shell procedures. Before describing these structures, a few terms need to be defined.

A *simple command* is any single irreducible command specified by the name of an executable file. I/O redirection arguments can appear in a simple command line and are passed to the shell, not to the command.

A *command* is a simple command or any of the shell control commands described below.

A *pipeline* is a sequence of one or more commands separated by vertical bars (|). In a pipeline, the standard output of each command 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, making the shell wait for the pipeline to finish before reading the next pipeline. On the other hand, the ampersand (&) causes asynchronous background execution of the preceding pipeline, allowing sequential and background execution. A background pipeline continues execution until it terminates.

Other uses of the ampersand include off-line printing and background compilation. 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 sending the interrupt or quit signals. The interrupt signal is usually sent by typing the DEL or BREAK key. The quit signal is usually sent by typing a CONTROL-\.

CONTROL-D will log you out and abort the command. To prevent this from happening, use the **nohup** command to make the command immune to hangups and logouts. If the preceding example did not contain **nohup** and if you log out while **cc** is still executing, **cc** will be killed and your output will disappear.

The ampersand operator should be used with restraint, especially on heavily-loaded systems. The work of other users will be slowed down if you run a large number of background processes.

The and-if and or-if (&& and ||) operators cause conditional execution of pipelines. Both of these are of equal precedence when evaluating command lines (but both are lower than the ampersand (&) and the vertical bar (|)). In the command line,

```
$ cmd1 || cmd2
```

the first command, **cmd1**, is executed and its exit status examined. Only if **cmd1** fails (that is, has a nonzero exit status) is **cmd2** executed.

The and-if operator (&&) yields a complementary test. For example, in the command line,

```
$ cmd1 && cmd2
```

the second command is executed only if the first succeeds (and has a zero exit status). In the sequence below, each command is executed in order until one fails.

```
$ cmd1 && cmd2 && cmd3 && ... && cmdn
```

A simple command in a pipeline may be replaced by a command list enclosed in either parentheses or braces. The output of all the commands so enclosed is combined into one stream that becomes the input to the next command in the pipeline. The following line formats and prints two separate documents.

```
${ nroff -mm text1; nroff -mm text2; } | lpr
```

Note that a space is needed after the left brace and that a semicolon must 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 word **fi** indicates the end of the **if** statement. (**fi** is **if** spelled backwards.) The example **cmd1 || cmd2** in the previous section can be rewritten using an **if** statement as follows.

```
if            cmd1
              test $? != 0
then          cmd2
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.

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** (else-if) clause, although the **case** statement 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
```

The previous example is executed as follows: if the value of the first positional parameter is a file name (**-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.

if commands may be nested (but be sure to end each one with a **fi**). The RETURNs 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 alternative notation for the **test** command uses brackets to enclose the expression being tested. For example, the previous example might have been written as follows.

```
if          [ -f "$1" ]                 #is $1 a file?
then        pr $1
elif        [ -d "$1" ]                 #else, is $1 a directory?
then        (cd $1; pr *)
else        echo $1 is neither a file nor a directory
fi
```

Note that a space after the left bracket and one before the right bracket are essential in this form of the syntax.

Using the case Statement

A multiple test conditional is provided by the **case** command. The basic format of the **case** statement is

> **case** *string* **in**
>   *pattern* **)** *command-list* **;;**
>   **...**
>   *pattern* **)** *command-list* **;;**
> **esac**

The shell tries to match *string* against each pattern in turn, using the same pattern-matching conventions as in file name generation. If a match is found, the command list following the matched pattern is executed; the double semicolon (;;) serves as a break out of the **case** and is required after each command list except the last. Note that only one pattern is ever matched, and that matches are attempted in order, so that if a star (*) is the first pattern in a **case**, no other patterns are looked at.

More than one pattern may be associated with a given command list by specifying alternative patterns separated by vertical bars (|). For example,

```
case $i in
            *.c)            cc $i;;
            *.h | *.sh)     : do nothing;;
            *)              echo "$i of unknown type";;
esac
```

No action is taken for the second set of patterns because the null, colon (:) command is specified. The star (*) is used as a default pattern, because it matches any word.

The exit status of **case** is the exit status of the last command executed in the **case** command. If no commands are executed, then **case** has a zero exit status.

Conditional Looping

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 RETURN 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

Often, one wishes to perform some set of operations for each file in a set of files, or execute some command once for each of several arguments. The **for** command can be used to accomplish this. The **for** command has the format

> **for** *variable* **in** *word-list* **do** *command-list* **done**

Here *word-list* is a list of strings separated by blanks. The commands in the *command-list* are executed once for each word in the *word-list*. The *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
```

The first occurrence of **CFILE** immediately after the word **for** has no preceding dollar sign, because 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 and execute it.

```
$ chmod + x echo2
$ echo2 ma pa bo fi yo no
mama
papa
bobo
fifi
yoyo
nono
```

Loop Control

The **break** command can be used to terminate execution of a **while** or a **for** loop. **continue** requests 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 (that is, 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, that is, 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
               "")continue ;;         # just a carriage return, keep on going
                *) echo $response ;;  # 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 CONTROL-D, which is the same as 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 0" at the end of the file.

Command Grouping

Two operators are used for grouping commands in the shell: parentheses and braces. Parentheses cause the shell to create a subshell that reads the enclosed commands. Both the right and left parentheses are recognized wherever they appear in a command line--they can appear as literal parentheses *only* when enclosed in quotation marks. For example, if you attempt to define a shell variable called **var** as **garble(stuff)**, you get an error message.

```
$ var = garble(stuff)
syntax error: '(' unexpected
```

You must quote the parentheses. In the XENIX operating system, preceding a character with a backslash has the same effect as enclosing it in single quotes. The following command lines have the same result.

```
$ var = garble'('stuff')'
$ var = 'garble(stuff)'
$ var = garble\(stuff\)
```

This capability of creating a subshell by grouping commands is useful when performing operations without affecting the values of variables in the current shell, or when temporarily changing the working directory and executing commands in the new directory without having to return to the current directory.

The current environment is passed to the subshell and variables 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 formatted and sent to the line printer.

Interpret the above commands this way. The first example sets the shell variable **CURRENTDIR** to the value of the **pwd** command (**pwd** stands for print-working-directory). Those are backquotes around the pwd, just like those enclosing the **date** command in a previous example. The semicolon separates two commands on the same line. The **cd** command changes the working directory to **/usr/docs/otherdir.** The **cd** command will not create that directory; it must already exist for you to change to it.

The next line invokes the **nroff** program on the file **doc.n.** Note that, in this case, no macro package is used. A previous **nroff** example using a macro package employed the flag **-mm.** The output of **nroff** is by default the terminal screen. The **| lpr** pipes the output to the line printer instead. (The **|** is the pipe symbol.) The ampersand puts the process in the background. That means you can enter additional shell commands before the **nroff** process has finished. This is often convenient because your terminal does not "go away" on you. The **nohup** option ensures that the process does not die if you log out. Finally, the second **cd** returns to your original working directory, whose pathname you saved in the shell variable **CURRENTDIR.**

The second example accomplishes the same result. Enclosing a series of commands in parentheses ensures that they are executed in a subshell spawned from the invoking shell. There is no need to save the pathname of the original shell. The second example automatically returns you to your original working directory. In the second example, blanks or RETURNs surrounding the parentheses are allowed but not necessary. When entering a command line at the terminal, the shell will prompt with the value of the shell variable PS2 if an end parenthesis is expected. PS2 is your secondary prompt, defined in your **.profile** file.

Braces ({ }) may also be used to group commands together. Both the left and the right brace are recognized only if they appear as the first (unquoted) word of a command. The opening brace may be followed by a RETURN (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. Refer to the **sh** entry in the *XENIX 286 Reference Manual* for more information. Pay special attention to the "Notes" section.

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 above. However, each command in a pipeline is run as a separate process 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:

- 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).

- 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

```
. process
```

causes the shell to read commands from *process* without spawning a new shell. Changes made to variables in *process* 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 your login shell by reading the **.profile** file with

```
$. .profile
```

Interrupt Handling

Shell procedures can use the **trap** command to disable a signal (cause it to be ignored), or redefine its action. The XENIX operating system provides a mechanism called signals to communicate with running processes. You have already come across the interrupt and quit signals. Typically you can send an interrupt signal by pressing the DEL key on your terminal. A signal is identified by a number. The number of the interrupt signal is 2. The form of the **trap** command is

trap *argument signal-list*

Here *argument* is a string to be interpreted as a command list and *signal-list* consists of one or more signal numbers. The commands in *argument* are scanned at least once, when the shell first encounters the **trap** command. Because of this, it is usually wise to use single rather than double quotation marks to surround these commands. Single quotation marks inhibit immediate command and variable substitution. This becomes important, for instance, when one wishes to remove temporary files and the names of those files have not yet been determined when the **trap** command is first read by the shell. The following procedure will print the name of the current directory in the 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
do
        cd $i
        ls -l  # commands to be executed in directory $i here
done
```

For the above shell script to work correctly the directories listed in the **for** statement must exist and be accessible. The command **ls -l** is included so that the shell script puts output on the screen.

Beware that the same procedure with double quotation marks causes the shell to print the name of the directory from which the procedure was first executed.

```
(trap "echo `pwd` >errdirect" 2 3 15)
```

A memory allocation signal (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 *argument* 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 *argument* 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 ensure that temporary files are removed upon termination of a procedure. Consider the following example, first without a **trap** command. Be sure to create a directory called temp in your home directory. Use the **mkdir** command. (It stands for make directory.)

```
$ mkdir tmp
```

Then, construct an executable file with the following two lines. The first line sets a shell variable to a pathname for a file under **tmp.** **$HOME** is a shell variable defined in **.profile** and contains the path name of your login directory. **$$** is the symbol for the shell process id number. This number is a convenient one to use when you want a name that stands a reasonable chance of being unique.

```
temp = $HOME/tmp/$$
ls > $temp        #commands that use $temp here
```

The result of this script is that you create a file in **tmp** that contains the names of the files in your working directory. The file's name is some number, the process id that the script got when it ran.

Now, assume that you want to delete that temporary file (it's not temporary unless you delete it) when you complete the script under certain conditions, namely receiving one of a list of signals. Edit the script to look as follows.

```
temp = $HOME/tmp/$$
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 (kill) 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 0** 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. When you execute the modified version of the script and look in the directory **tmp**, you will not find a new file created.

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 (CONTROL-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; eval $x; done
          fi
done
```

Here's an example of the use of the above shell script. Assume that it is in an executable file called **scan.** Assume that you execute **scan** from your login directory, **/usr/ted,** and that this directory contains the directories, **bin, dir1,** and **junk.** The directory **bin** contains one file called **s1,** and the directory called **junk** is empty.

```
$scan
bin:
pwd
/usr/ted/bin
bin:
ls -l
total 0
-rw-r--r--    1  ted     xenix     40  Aug 13 17:01  s1
bin:
enter a CONTROL-D
dir1:
enter a CONTROL-D
junk:
ls -l
total 0
junk:
enter a CONTROL-D
$
```

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** with no arguments. For example,

```
$ trap 'echo hello' 2 15
$ trap
2: echo hello
15: echo hello
```

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 are 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

The shell contains several internal special commands, some of which have already been mentioned. The shell does not fork to execute these commands, so no additional processes are spawned. These commands should be used whenever possible, because they are, in general, faster and more efficient than other XENIX commands. 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. Descriptions of the remaining special commands are given here.

:	The null command. This command does nothing and can be used to insert comments in shell procedures. Its exit status is zero (true). Its utility as a comment character has largely been supplanted by the number sign (#), which can be used to insert comments to the end-of-line. Note that 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 *argument*	Make *argument* the current directory. If *argument* is not a valid directory, or the user is not authorized to access it, a nonzero exit status is returned. Specifying **cd** with no *argument* is equivalent to typing **cd $HOME**, which takes you to your home directory.
exec *argument ...*	If *argument* is a command, then the shell executes the command without forking and returning to the current shell. This is effectively a "goto" and no new process is created. Input and output redirection arguments are allowed on the command line. If *only* input and output redirection arguments appear, then the input and output of the shell itself are modified accordingly.

newgrp *argument...*	The **newgrp** command is executed, replacing the shell. **newgrp** in turn creates a new shell. Note that 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 *variable...*	One line (up to a RETURN) is read from the standard input and the first word is assigned to the first variable, the second word to the second variable, and so on. All words left over are assigned to the last variable. The exit status of **read** is zero unless an end-of-file is read.
readonly *variable...*	The specified variables are made read-only 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	The accumulated user and system times for processes run from the current shell are printed.
umask *nnn*	The user file creation mask is set to *nnn*. If *nnn* is omitted, then the current value of the mask is printed. This bit-mask is used to set the default permissions when creating files. For example, an octal **umask** of 137 corresponds to the following bit-mask and permission settings for a newly created file.

User	user	group	other
octal	1	3	7
bit-mask	001	011	111
permissions	rw-	r--	---

See **umask** in the *XENIX 286 Reference Manual* for information on the value of *nnn*.

ulimit *nnn*	The process file size limit is set to *nnn*; the value of *nnn* is in units of 512K-byte blocks.
wait	The shell waits for all currently active child processes to terminate. The exit status of **wait** is always zero.
eval *argument*	Arguments are read as input to the shell and the resulting command(s) executed.

Creating and Organizing 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

```
process argument
```

rather than

```
sh process argument
```

The second step may be omitted for a procedure to be used once or twice and then discarded, but is recommended for frequently-used ones. 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 might then be invoked from within the current directory by typing

```
$ mailall letter joe bob
```

Here **letter** is the name of the file containing the message you want to send, and **joe** and **bob** are people you want to send the message to. Note that shell procedures must always be at least readable, so that the shell itself can read commands from the file.

If **mailall** were thus created in a directory whose name appears in the user's PATH variable, the user could change working directories and still use the **mailall** command.

Shell procedures may be created dynamically. A procedure may generate a file of commands, invoke another instance of the shell to execute that file, and then remove it. An alternative approach is that of using the **dot** command (.) to make the current shell read commands from the new file, allowing use of existing shell variables and avoiding the spawning of an additional process for another shell.

Many users prefer writing shell procedures to writing C programs. This is true for several reasons.

- A shell procedure is easy to create and maintain because it is only a file of ordinary text.
- A shell procedure has no corresponding object program that must be generated and maintained.
- A shell procedure is easy to create quickly, use a few times, and then remove.
- 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 containing commands and shell procedures are named **bin.** The name **bin** is derived from the word "binary" and is used because compiled and executable programs are often called "binaries" to distinguish them from 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 directories.

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 that 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 but not execute the commands in the procedure. Using **set -nv** at the beginning of a file will accomplish this.

-k This flag causes all arguments of the form *variable=value* to be treated as keyword parameters. When this flag is not set, only such arguments that appear before the command name are treated as keyword parameters.

Supporting Commands and Features

Shell procedures can make use of any XENIX command. The commands described in this section are either used frequently in shell procedures or explicitly designed for such use.

Conditional Evaluation

The **test** command evaluates the expression specified by its arguments and, if the expression is true, **test** returns a zero exit status. Otherwise, a nonzero (false) exit status is returned. **test** also returns a nonzero exit status if it has no arguments. Often it is convenient to use the **test** command as the first command in the command list following an **if** or a **while.** Shell variables used in **test** expressions should be enclosed in double quotation marks if they are possibly 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

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; the *XENIX 286 Reference Manual* contains a complete list of the conditional evaluation options.

-r *file*	True if the named file exists and is readable by the user.
-w *file*	True if the named file exists and is writeable 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 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 above options may be combined with the following operators.

!	Unary negation operator.
-a	Binary logical AND operator.
-o	Binary logical OR operator; it has lower precedence than the logical AND operator (-a).
( )	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, file names, etc. are separate arguments to **test.**

Echoing Arguments

The **echo** command has the following syntax.

> **echo** [*options*] [*arguments*]

echo copies its arguments to the standard output, each followed by a single space, except for the last argument, which is normally followed by a RETURN. Often it is 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 may cause an error or system crash.

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 RETURN from the end of the echoed line. The following two commands prompt for input and enable typing on the same line.

```
echo -n 'enter name:'
read name
```

The **echo** command also recognizes several escape sequences described in **echo** in the *XENIX 286 Reference Manual.*

Expression Evaluation

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 backquotes to set a variable. The most common uses of **expr** are counting iterations of a loop and in using its pattern-matching capability to pick apart strings. Some typical examples follow. Create an executable file with the following lines in it. You can name the file anything you want, but this example assumes that you named it **expeval.**

```
a=2                      # define the variable a
echo a=$a
A=`expr $a + 1`          # add one to a and call the answer A
echo A=$A
substring=`expr "$1" : '..\(.*\)' `     # put third thru last characters of $1
                                        # into substring
echo $substring
c=`expr "$1" : '.*' `                   # obtain length of $1
echo c=$c
```

Now, execute the script **expeval** with the argument **123456.** The output looks as follows.

```
$ expeval 123456
a=2
A=3
substring=3456
c=6
```

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 the shell receives an interrupt signal. With most systems, you can provide an interrupt signal by pressing the DEL key.

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 to *command* until a line is read consisting only of *eofstring*. (By appending a minus sign to the input redirection symbol (<<-), leading tabs are deleted from each line of the input document before the shell passes the line to *command*.)

The shell creates a temporary file containing the input document and performs variable and command substitution on its contents before passing it to the command. Pattern-matching on file names is performed on the arguments of command lines in command substitutions. To prohibit all substitutions, you may quote any character of *eofstring*. Remember from the example in the section called "Command Grouping" that quoting a character (enclosing it in single quotes) is equivalent to preceding it with a \.

```
command  <<  \eofstring
```

Here is an example of how the backslash in *eofstring* may be useful. Notice that after the **cat** command and before you enter *eofstring*, you see your secondary prompt. You can define your secondary prompt by setting PS2 in your **.profile**, but be aware that your new PS2 does not take effect until you log in again or until you source **.profile** with the **dot** (.) command, as follows: **. .profile.** This example assumes that your secondary prompt is >.

```
$ a = value
$ cat  <<\xx
> hello
> $a
> xx
hello
$a
$
```

Without the backslash, you get the value of $a, not the string $a. By the way, you could have entered **cat <<x\x** as well as **cat <<\xx.**

```
$ a = value
$ cat  <<xx
> hello
> $a
> xx
hello
value
$
```

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 removed
xx
```

This in-line input document feature is most useful in shell procedures. Note that in-line input documents may not appear within backquotes.

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, you can associate output with any file descriptor by using the **write** system call (see the *XENIX 286 C Library Guide*). 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, you can direct output normally 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.

Consider the following. If you try to delete a nonexistent file, you get an error message sent to file descriptor 2. By default, this message appears on the screen. For example,

```
$ rm baloney
rm: baloney non-existent
```

Assume that you want to redirect that error message to a file instead. Then, issue the command as

```
$ rm baloney 2>errfile
```

The file **errfile** is created and it contains the error message. You can display this file with the **cat** command. Now assume that you've redirected file descriptor 1 to a file called **otherfile** and you want to redirect file descriptor 2 to that same file. Issue the command as follows.

```
$ rm baloney 1>otherfile 2>&1
```

The order here is significant: first, file descriptor 1 is associated with **otherfile**; 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 **otherfile**, because at the time of the error output redirection, file descriptor 1 would still have been associated with the terminal.

This mechanism can also 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, dependent 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 any one 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. "Parameter" as used in the following definitions refers to either a digit or a variable name.

${*variable:-string*}	If *variable* is set and is not null, 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 not null, then substitute the value $*variable* in place of this expression. Otherwise, set *variable* to *string*, and then substitute the value $*variable* in place of this expression. Positional parameters may not be assigned values in this fashion.

${*variable*:+*string*} If *variable* is set and is not null, 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.

${*variable*:?*string*} If *variable* is set and is not null, 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.

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 shell script illustrates the above rules for conditional substitution. The part of the line after the # is a comment that explains what happens on that line.

```
echo 'testing ${variable:-string}'
echo d = $d                     #you see that d is null
echo ${d:-hello}                #you see hello
echo d = $d                     #note that the value of d is not changed
d = 'I am set and not null'     #setting d
echo ${d:-hello}                #you see d's value
d = ''                          #then set d back to null

echo 'testing ${variable: = string}'
echo d = $d                     #you see that d is null
echo ${d: = hello}              #you see hello
echo d = $d                     #note that the value of d is changed
d = ''                          #then set d back to null

echo 'testing ${variable: + string}'
d = 'I am set and not null'
echo ${d: + hello}              #you see hello
echo d = $d                     #note that d is unchanged
d = ''                          #then set d back to null

echo 'testing ${variable:?string}'
echo d = $d                     #you see that d is null
echo ${d:?messagestring}        #you see the message string
```

The two examples below further illustrate the use of this facility.

- This example performs an explicit assignment to the PATH variable.

  ```
  "PATH"=${PATH:-':/bin:/usr/bin'}
  ```

  If PATH has ever been set and is not null, then keep its current value; otherwise, set it to the string ":/bin:/usr/bin".

- 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 may be specified on the command line when invoking the shell and 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. Your login shell 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. Double quotation marks should be used to enclose a multiword string 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 should not worry about optimizing shell procedures unless they are intolerably slow or are known to consume an inordinate amount of a system's resources.

The same kind of iteration cycle should be applied to shell procedures as to other programs: write code, measure it, and optimize only the *few* important parts. You 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 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 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, you should know which commands are built into the shell and which are not. Here is the alphabetical list of built-in commands.

break	case	cd	continue	eval	exec	exit
export	for	if	newgrp	read	readonly	set
shift	test	times	trap	umask	until	wait
while	.	:	{}			

Parentheses are built into the shell, but commands enclosed within them are executed as a child process, i.e., the shell does a **fork**, but no **exec**. Any command not in the above list requires both **fork** and **exec**.

You should always have an estimate of the number of processes generated by a shell procedure. In the bulk of observed procedures, the number of processes created (not necessarily simultaneously) can be described by

processes = (*k***n*) + *c*

where *k* and *c* are constants, and *n* may be the number of procedure arguments, the number of lines in some input file, the number of entries in some directory, or some other obvious quantity. Efficiency improvements are most commonly gained by reducing the value of *k*, sometimes to zero.

Any procedure whose complexity measure includes n^2 terms or higher powers of *n* is likely to be intolerably expensive.

As an example, here is an analysis of a procedure named **split**, whose text is given below.

```
#                   split
trap 'rm temp$$; trap 0; exit' 0 1 2 3 15
start1=0 start2=0
b='[A-Za-z]'
cat > temp$$
                                # read stdin into temp file
                                # save original lengths of $1, $2
if test -s "$1"
then start1=`wc -l < $1`
fi
if test -s "$2"
then start2=`wc -l < $2`
fi
grep "$b" temp$$ >> $1
                                # lines with letters onto $1
grep -v "$b" temp$$ | grep '[0-9]' >> $2
                                # lines with only numbers onto $2
total="`wc -l < temp$$`"
end1="`wc -l < $1`"
end2="`wc -l < $2`"
lost="`expr $total - \(nd1 -tart1\) \ -\(nd2 - tart2\)`"
echo "$total read, $lost thrown away"
```

For each iteration of the loop, there is one **expr** plus either an **echo** or another **expr.** One additional **echo** is executed at the end. If *n* is the number of lines of input, the number of processes is 2*n+1.

Some types of procedures should *not* be written using the shell. For example, if one or more processes are generated for each character in some file, it is a good indication that the procedure should be rewritten in C. Shell procedures should not be used to scan or build files a character at a time.

Number of Data Bytes Accessed

Whenever you can, reduce the number of processes running; XENIX is more efficient when it spends time passing data rather than creating many small processes. Some filters shrink output, others increase it. Put the "shrinkers" first when the order is irrelevant. For instance, the second example below is faster because the input to **sort** will be much smaller.

```
sort file | grep pattern
grep pattern file | sort
```

Shortening Directory Searches

Directory searching can consume a great deal of time, especially in those applications that use deep directory structures and long path names. Judicious use of **cd,** the **change directory** command, can help shorten long path names 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 path name that precedes the needed path name 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
```

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 example 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

Avoid excessively large directories. 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 cumbersome 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 because the directory is still the same size.

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.

- Place the procedure in a file with the indicated name.
- Give the file execute permission with the **chmod** command.
- Move the file to a directory where commands are kept, such as your own **bin** directory.
- Make sure that the path of the **bin** directory is specified in the PATH variable found in **.profile.**
- Execute the named command.

binuniq

```
ls /bin /usr/bin | sort | uniq -d
```

This procedure determines which files are in both **/bin** and **/usr/bin.** The files in **/bin** will override those in **/usr/bin** during most searches, and duplicates need to be deleted. 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 related positional parameters. 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' '  12'|sort -u |\
tr 'A-Z' 'a-z' | sort | uniq -d
```

This procedure is an example of the kind of process created by the left-to-right construction of a long pipeline. Note the use of the backslash at the end of the first non-comment program line as the line continuation character.

It may not be immediately obvious how this command works. You may wish to consult **tr, sort,** and **uniq** in the *XENIX 286 Reference Manual* if you are completely unfamiliar with these commands.

The **tr** command translates all characters except letters and digits into RETURN characters, and then produces repeated RETURN characters. This leaves each string (in this case, any contiguous sequence of letters and digits) on a separate line. The **sort** command sorts the lines and emits only one line from any sequence of one or more repeated lines. The next **tr** converts everything to lowercase, so that identifiers differing only in case become identical. The output is sorted again to bring such duplicates together. The "uniq -d" prints (once) only those lines that occur more than once, yielding the desired list.

The process of building such a pipeline relies on the fact that pipes and files can usually be interchanged. The first line below is equivalent to the last two lines, assuming that sufficient disk space is available.

```
cmd1 | cmd2 | cmd3

cmd1 > temp1; < temp1 cmd2 > temp2; < temp2 cmd3
rm temp[123]
```

Starting with a file of test data on the standard input and working from left to right, each command is executed, taking its input from the previous file and putting its output in the next file. The final output is then examined to make sure that it contains the expected result. The goal is to create a series of transformations that will convert the input to the desired output.

Although pipelines can give a concise notation for complex processes, you should exercise some restraint, since such practice often yields incomprehensible code.

draft

```
#     Usage: draft file(s)
#     Print manual pages for Diablo printer.
for i in $*
      do nroff -man | lpr
done
```

Users often write this kind of procedure for convenience in dealing with commands requiring distinct flags that cannot be given default values 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 given to it as an argument. It places any field that begins with a colon on a new line. Thus, if given the input,

```
joe newman:   13509 NE 78th St:   Redmond, Wa 98062
```

listfields produces

```
joe newman
13509 NE 78th St
Redmond, Wa 98062
```

In the command line, note the use of the **tr** command to convert colons to linefeeds.

mkfiles

```
#     Usage:   mkfiles  pref  [quantity]
#     makes  "quantity"  files,  named  pref1,  pref2,  ...
#     default  is  5  as  determined  on  following  line.
quantity=${2-5}
i=1
while  test  "4i"   =le   "$quantity"
do
      >$1$i
      i="`expr  $i   +   1"
done
```

The **mkfile** procedure uses output redirection to create zero-length files. The **expr** command counts 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 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

To determine which files in a directory contain only text, **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 that silently tests whether any of the files in the argument list are text files.

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.

Metacharacters and Reserved Words

Syntactic

|	Pipe symbol
&&	And-if symbol
||	Or-if symbol
;	Command separator
;;	Case delimiter
&	Background commands
( )	Command grouping
<	Input redirection
<<	Input from a document
>	Output creation
<	Output append
#	Comment to end-of-line

Patterns

*****	Match any character(s) including none
?	Match any single character
[...]	Match any of enclosed characters

Substitution

${...}	Substitute shell variable
\`...\`	Substitute command output

Quoting

\fP	Quote next character as literal with no special meaning
'...'	Quote enclosed characters except the back quotation marks (\`)
"..."	Quote enclosed characters except $ \` "

Reserved words

{ }
case
elif
else
do
done
esac
fi
for
if
in
then
until
while

intel®

CHAPTER 4
ed: A LINE-ORIENTED TEXT EDITOR

Introduction

ed is a general-purpose, full-screen, line-oriented editor used to create and modify text. Rather than entering a stream of revisions to be made as in **sed**, or moving the through the file as in **vi**, editing in **ed** is done by specifying a line number to be edited and then making the revisions to that line.

Basic Concepts

This section introduces the basic concepts of **ed**.

Entering and Exiting ed

To invoke **ed**, type

```
ed filename
```

where *filename* is the name of a new or existing file. When you enter **ed**, it prompts you for commands with an asterisk (*). To create or append text, see "Creating and Appending Text" later in this chapter.

To exit the editor, type

```
q
```

If you try to exit the editor without saving any changes to a file, **ed** returns the following warning, telling you that the changes have been made and the file hasn't been saved:

```
warning: expecting 'w'
```

If you still want to exit without saving the changes, type another **q**. In most cases you will want to exit by typing

```
w
q
```

The **w** command (discussed in detail later in this chapter) saves the changes and enables you to type **q** to quit **ed**.

Line Numbers

Any time a command changes the number of lines in the editing buffer, **ed** renumbers the lines. At all times, every line in the editing buffer has a line number. Many editing commands will take either single line numbers or line number ranges as prefixing arguments. These arguments normally specify the lines in the editing buffer to be affected by the given command. The current line can be specified with a special line number called "dot" and is represented by a period (.). You can determine the actual line number of the current line by entering

```
.=
```

The Editing Buffer

Each time you invoke **ed**, an area in the memory of the computer is allocated where you will perform all editing operations. This area is called the "editing buffer." When you edit a file, a copy of the original is placed in the editing buffer, where you will work on it. Only when you write out the buffer (using the **w** command) do you change the original file.

Calling a File

The edit command (**e**) places the entire contents of a file into the editing buffer. During a single **ed** session you may need to edit a number of files; **e** enables you to read files into the buffer without exiting **ed**. Once you enter **ed**, you can edit a file, write out the buffer by using **w**, and read in the next file to edit by entering the command

e *filename*

The specified file is read into the buffer and **ed** displays the number of characters in the file. Any data in the buffer is deleted before the new file is read in.

If you use **e** to read a file into the buffer, you don't need to use a file name after a subsequent **w** command; **ed** remembers the last file name used in an **e** command, so **w** automatically writes to this file.

Writing Out the Editing Buffer

You will probably want to save your text for later use. To write out the contents of the buffer into a file, use the **write (w)** command followed by the name of the file you want to write to. This copies the contents of the buffer to the specified file, destroying any previous contents of the file. Leave a space between **w** and the file name. For example, to save the buffer in a file named **text**, type

```
w text
```

ed responds by printing the number of characters in the text. (Blanks and the RETURN character at the end of each line are included in the character count.) Writing out a file makes a copy of the text in the editing buffer--the buffer's contents are not disturbed, so you can continue editing it. When you invoke **ed** by using a *filename* argument, a **w** command by itself writes the buffer out to *filename*.

ed always works on a copy of a file, not the file itself. The contents of a file remain unchanged until you write out the editing buffer. Writing out the editing buffer occasionally as you work is an excellent safety measure; if the system crashes all the text in the buffer is lost, but any text written out is usually safe.

Writing Out Part of a File

During an editing session you may need to write out only part of the editing buffer. For example, you may want to split a table out into a separate file so it can be formatted and tested separately. Suppose that in the file being edited we have

```
.TS
[text for table]
.TE
```

which is the way a table is set up for the **tbl** program. To isolate the table in a separate file called **table**, find the start of the table (the ".TS" line) and then write out the text of the table.

First type

```
/^.TS/
```

This prints out the line

```
.TS
```

Next, type

```
.,/^.TE/w table
```

which means "write out from the current line (.) to the line beginning with .TE and put it in the file called **table.**"

If you are confident, you can do it all at once with

```
/^.TS/;/^.TE/w table
```

By using the **w** command, you can write out a group of lines instead of the whole file. In fact, you can even write out a single line; just give one line number instead of two.

Changing File Name to Write Out to

The **ed** file command (**f**) enables you to determine the name of the last file written to and specify a new file name for the editing buffer to write to. Check the name of the last file written to by entering **f** at the **ed** prompt (*).

To change the file name that the editing buffer should write to, enter **f** in the following format:

```
f newfilename
```

Changing the file name can be very useful for documents such as contracts, where certain parts of the text may change but the original file serves as a master form and the new file is a record of the individual transaction. You would invoke **ed** and specify the original file name, use **f** to specify a new file name, edit the contents of the buffer, and write out the revised text to the new file name. If you needed to make changes to the copy of the individual transaction, you could use the same process, ensuring that a record of each change would be saved.

Commands

ed prompts for commands with an asterisk (*). Enter commands by typing them at the keyboard and then pressing RETURN. Most commands are single characters that can be preceded by a line number or a line number range. By default, most commands operate on the current line (.). Many commands take file name or string arguments used by the command when it is executed.

Undoing Commands

Occasionally you will make a substitution in a line only to realize too late that it was a mistake. The undo (**u**) command enables you to "undo" the most recent command. Thus, the last line that was substituted can be restored to its previous state by typing

```
u
```

This command does not work when global commands are used in combination with the substitution command.

Displaying Lines

The print (**p**) command displays the contents of the editing buffer. By using line numbers, you can display the entire buffer or just portions of it. The **p** command format is

*beginning line number, ending line number*p

Suppose you want to print the entire buffer; you could use "1,3p" as above if you know there are exactly three lines in the buffer. Since it is unlikely that you know how many lines are in the buffer, ed provides a shorthand symbol for the line number of the last line in the buffer--the dollar sign (**$**). Use it this way:

```
1,$p
```

This will print the entire buffer (from line 1 to the last line). If you want to stop the printing before it is finished, press the DELETE key. **ed** then displays

```
?
interrupt
```

and waits for the next command.

To print the last line in the buffer, type

```
$p
```

You can print any single line, including the current line (.), by typing the line number followed by **p**. Typing

```
2p
```

displays the second line of the buffer. In fact, you can abbreviate even further by deleting the **p** from the command line; at the command

ed prints the current line. You can delete the **p** only when you are specifying a single line. If you want to display multiple lines, you must include the **p**. The current line may also be displayed by entering

```
p
```

The next step is to use address arithmetic to combine the line numbers like dot (.) and dollar sign ($) with plus (+) and minus (-). The command

```
$-1
```

literally means "print the last line of the buffer ($) minus one (-1)"; in other words, the next to the last line in the buffer. For example, to view the last six lines of the buffer when you don't know how many lines are in the buffer, type

```
$-5,$p
```

If the file has less than six lines you'll get an error message. Entering the command

```
.-3,.+3p
```

prints seven lines: the three lines before the current line, the current line, and the three lines after the current line.

You can also use plus and minus as line numbers by themselves. For example, entering

```
-
```

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. Typing

```
---
```

moves back three lines, as does

```
-3
```

Displaying Tabs and Control Characters

ed provides two commands for printing the contents of the text you are editing: **p** (described in the previous section) and **l** (list).

The **l** command makes visible those characters normally invisible, such as tabs. If you list a line that contains tabs, **l** prints each tab as ">". This makes it much easier to correct extra spaces adjacent to tabs.

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 the screen.

Occasionally the **l** command will print a string of numbers preceded by a backslash, such as \7 or \16. These combinations are used to make visible those characters that normally don't print, like formfeed, vertical tab, or bell. Each backslash-number combination represents a single ASCII character. (Note that numbers are octal and not decimal.) Such characters may cause unpredictable behavior when printed on some terminals. Since they are rarely used, their presence usually indicates an error.

Interrupting ed

If you press the DELETE key while **ed** is executing a command, your file is restored (as much as possible) to what it was before the command began. However, some changes are irrevocable. If you are reading in or writing out a file, making substitutions, or deleting lines, these functions will be stopped in some unpredictable state, so it is unwise to stop them. Dot may or may not be changed.

When using the print command, dot is not changed until the printing is done. If you print to a certain line and press DELETE to stop printing, dot will not be set to that line; it is left where it was when the **p** command was started.

Escaping to the Shell

Sometimes you may need 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*

where *command* is a shell command, your current editing state is suspended, and the specified XENIX command is executed. When the command finishes, **ed** will signal you by printing another exclamation point (!); at that point you can resume editing.

Creating and Appending Text

Suppose that you want to create some text starting from scratch. This section shows you how to create text in a file. Later we'll talk about how to edit existing files.

When you first invoke **ed**, it is like working with a blank piece of paper--there is no text or information present. These must be supplied by the person using **ed**, usually by typing in the text or by reading it in from a file. We will start by typing in some text and discuss how to read files later.

The first command we will discuss is **append (a)**, entered as the letter "a" on a line by itself. It means "append (or add) text lines to the buffer as they are typed in." Appending is like writing new material on a piece of paper.

To enter lines of text into the buffer, just type an "a" at the **ed** prompt (*), followed by a RETURN, followed by the lines of text you want, like this:

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

To stop appending text, enter a period (.) or CONTROL-D on a blank line. If **ed** seems to be ignoring you, type an extra line with just a period on it.

After appending is completed, 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 **.** aren't there, because they are not text.

To add more text to what you already have, type another **a** command and continue typing your text.

If you make a mistake while entering a command or use an incorrect command, **ed** displays an error message preceded by a question mark.

Deleting Lines

The delete command (**d**) deletes lines of text from the buffer. The lines to be deleted are specified for **d** the same way as for **p**. Thus, entering the command

```
2,$d
```

deletes lines 2 through the end. You can delete a single line by entering the line number to be deleted, followed by **d**.

Searching

The search command (//) enables you to search through a file for a particular string; its syntax is

```
/pattern/
```

This is also called a "context search expression." The slash and question mark are the only characters you can use to delimit a context search. In their simplest form, all context search expressions are like this--a string of characters surrounded by slashes. Context searches are interchangeable with line numbers, so they can be used to find and print a desired line.

Searching is useful for finding a single occurrence of a pattern when you don't know what line it is on. Use the search command to find the pattern you need and then you can make the necessary editing changes.

For example, the command

```
/their/
```

will locate 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
```

"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.) If the given string of characters can't be found in any line, **ed** prints the error message

```
?
search string not found
```

The search command sets dot to the line where the pattern is found and prints the line for verification.

You can also search a file in reverse by using question marks instead of slashes. For example

```
?thing?
```

searches backward in the file for the word "thing" as does

```
?thing
```

This is especially handy when you realize that the string you want is prior to the current line.

Suppose the buffer contains the three lines

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

The ed commands

```
/Now/+1

/good/

/party/-1
```

are all context search expressions, and they all refer to the same line (line 2). 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 the pattern "the", and when the line is printed you discover that it isn't the "the" you wanted; you must repeat the search. However, you don't have to retype the search, because the construction

```
//
```

is a shorthand expression for "the previous pattern that was searched for". This can be repeated as many times as necessary. You can also go backward, since

```
??
```

searches for the same pattern, but in the reverse direction.

Searching with the 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. If there had been a line with a "b" in it before the "ab" line, the print command would return an error since the second line number would be less than the first and you can't print lines in reverse order.

This happens 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 (;) may be used just like the comma, except the 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 "a" is found, dot is set to that line, and then "b" is searched for starting at dot + 1.

Suppose you want to find the second occurrence of "thing". You could type

```
/thing/
```

to find the first occurrence, then type

```
//
```

to find the second. However, you could find just the second by typing

```
/thing/;//
```

This says "find the first occurrence of 'thing', set dot to that line, then find the next occurrence and print only that."

Closely related is searching for the second-to-last occurrence of a pattern, as in

```
?thing?;??
```

To find the first occurrence of a pattern in a file, starting at an arbitrary place within the file, use the command

```
0;/thing/
```

which starts the search at line 1. This is one of the few places where 0 is a legal line number.

Searching and Replacing

This section discusses the change (**c**) command, used to change or replace one or more lines, and the insert (i) command, 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 period 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 any number of replacement lines.) 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.

The insert (i) command is similar to the **append** command. The syntax for the insert command is

```
i
text to be inserted
.
```

The search command can be used in conjunction with i to locate the place to insert text. For example

```
/string/i
text to be inserted
.
```

searches for the line containing the specified string and inserts the given text ahead of that line. If no line number is specified, the current line is used and dot is then set to the last line inserted.

Substituting Text

One of the most important **ed** commands is the substitute (**s**) command. This command is used to change phrases, words, or letters and correct spelling mistakes and typing errors.

The syntax for the substitute command is

starting-line,ending-lines/pattern/replacement/

pattern is replaced by *replacement,* 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 and the following error message is returned:

```
?
no match
```

Suppose, because of a typing error, line 4 reads

```
Now is th time
```

Use **s** to correct the misspelling of "the" by entering

```
4s/th/the/
```

This substitutes the characters "the" for the characters "th" in line 4. If no line numbers are given, the **s** command assumes you mean "make the substitution 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. If it didn't, you can try again. (With few exceptions, **p** can follow any command; no other multiple command lines are legal.)

You can also 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

```
Nowww is the time
```

you could type

```
s/ww//p
```

to produce

```
Now is the time
```

Note that two adjacent slashes mean "no characters", not a space.

You can also use // as the left side of a substitute command, to mean "the most recent pattern." For example, if you type

```
/country/
```

ed prints the line containing "country." If you now type

```
s//continent/p
```

this changes "country" to "continent".

So far, the substitution patterns demonstrated have only changed the first occurrence in each line. To change all occurrences of a pattern, you must add a **g** (for global) to the end of the substitution command. The section "Performing Global Commands" later in this chapter discusses the use of global commands in detail.

Metacharacters

Metacharacters are a set of special characters used to describe patterns of text in search and substitute commands. These patterns are called "regular expressions" and occur in several other important XENIX commands and utilities, including **grep** and **sed** (see the *XENIX 286 Reference Manual*). A complete list of the metacharacters follows:

```
\    .    ^    $    *    []    &
```

The following sections describe how to use these metacharacters in search and substitute commands.

Backslash

The backslash (\) turns off any special meaning that the following character has. As an example, the sequence

```
\.
```

in a search or substitute command changes the meaning of the period from "match any single character" to a literal period. When you are adding text with **a**, **i**, or **c**, the backslash has no special meaning.

Period

When used in a search or on the left-side expression of a substitute command, the period stands for any single character (this is frequently called a "wildcard" character). For example, the search command

```
/x.y/
```

finds any line where "x" and "y" occur separated by any single character, as in

```
x+y
x-y
x y
xzy
```

If you use a period in the right-side expression of a substitute command, the period assumes its literal meaning.

Caret

The caret (^) signifies the beginning of a line; the first character of every line is a caret (although it is invisible). To search for or substitute an expression at the beginning of a line, precede the expression with a caret. The search and substitute commands have the following formats when using the caret:

/^*expression*/

s/^*expression*/*expression*/

For example, suppose you are looking for a line that begins with "the". If you simply type

```
/the/
```

you will probably find several lines containing "the" before arriving at the one you want. But with

```
/^the/
```

you narrow the search context to "the" at the beginning of a line only, and thus arrive at the desired line more quickly.

The caret (^) also enables you to insert characters at the beginning of a line. For example

```
s/^/ /
```

places a space at the beginning of the current line.

Dollar Sign

The dollar sign ($) signifies the end of a line; the last character of every line is a dollar sign (although it is invisible). To search for or substitute an expression at the end of a line, follow the expression with a dollar sign. The search and substitute commands have the following formats when using the dollar sign:

```
/expression$/

s/expression$/expression/
```

For example, suppose you are looking for a line that ends with "the". If you simply type

```
/the/
```

you will probably find several lines containing "the" before arriving at the one you want. But with

```
/the$/
```

you narrow the search context to "the" at the end of a line only, and thus arrive at the desired line more quickly.

The dollar sign ($) also enables you to insert characters at the end of a line. For example, this places a period at the end of the current line:

```
s/$/./
```

Star

The star is useful for finding numerous occurrences of a single character. Literally, it means "find any number of consecutive occurrences, including zero, of the character that preceded the star followed by any designated text." For example, the command

```
/n*o
```

finds any number of occurrences of the letter "n" (including zero occurrences) followed by the letter "o." When using the star in substitution command lines, be very careful; if you specify the star with the wrong context, the file may be incorrectly changed.

Suppose that the line you want to edit is

text x.................y *text*

If you type

```
s/x.*y/x  y/
```

the result is unpredictable. If no other x's or y's occur on the line, the substitution can work, but not necessarily. The period matches any single character, so ".*" matches as many single characters as possible, and unless you are careful, it can remove more of the line than you expected.

Brackets

Brackets are used in search and substitution commands to specify a "character class". 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".

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 may appear within a character class, and only three special characters (^,], and -) appear inside the brackets; even the backslash doesn't have a special meaning. To search for any special characters, for example, type

```
/[.\$^[]/
```

Digits can be abbreviated as [0-9]; similarly, [a-z] stands for the lowercase letters, and [A-Z] for uppercase.

Ampersand

The ampersand (&) can be used on the right side of a substitute command to signify the string of text found on the left side of a substitute command. Using the ampersand eliminates repeating the string on both sides of the command and is extremely useful when adding text within a line. The syntax of the ampersand command is

s/*string*/& *new string*/

For instance, if you are editing the line

```
Now is the times
```

and want to change it to read

```
Now is the best of times
```

you would enter the command

```
s/the/& best of/
```

Here, the ampersand stands for "the", so that "the" is changed to "the best of".

The ampersand may be used more than once within the same command line; it always signifies the string on the left side of the substitution.

Performing Global Commands

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/^[0-9]/p
```

prints all lines in a file that begin with a number (numbered lists for instance).

The **v** command is identical to **g**, except that it operates on those lines that do not contain an occurrence of the pattern. For example, this prints all lines that don't begin with a number.

```
v/^[0-9]/p
```

Any command can follow **g** or **v**. The following command deletes all lines beginning with a number:

```
g/^[0-9]/d
```

The following command deletes all empty lines:

```
g/^$/d
```

One of the most useful commands that may be used in combination with the global command is the substitute command. For example, you could change the first occurrence of "Xenix" on every line to "XENIX" by typing

```
g/Xenix/s//XENIX/
```

If you wanted to change every occurrence of "Xenix" to XENIX", you would type

```
g/Xenix/s//XENIX/g
```

The command that follows a **g** or **v** command functions the same as a command on a line by itself. For example

```
g/^$/+
```

prints the line that follows each blank line (usually the first line in a paragraph). Plus (+) means "one line past dot".

The command

```
g/^Item 2/+,/^Item 8/-p
```

prints all lines that come just after the line beginning with "Item 2" and just before the line beginning with "Item 8". Minus (-) means one line before dot.

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. For example, this prints all lines between 1 and 10 that have the word "Item":

```
1,10g/Item/p
```

You can give more than one command under the control of a global command. For example, the following command will find all lines that contain "thing" and change the first occurrence of "a" on the line to "b" and the first occurrence of "x" on the line to "y":

```
g/thing/s/x/y/\
s/a/b/
```

The backslash (\) signals the **g** command that the commands continue on the next line; the **g** command terminates on the first line not ending with a backslash.

You can also 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 two lines of ten asterisks before each line beginning with "Item", type

```
g/^Item/i\
**********\
**********
```

There is no need for a final line containing a period (.) to terminate the **i** command, unless further commands are to be executed under the global command.

Copying Lines

ed provides a command, **t** (for transfer), for copying a group of one or more lines. This is often easier than writing and reading. The syntax for **t** is

starting-line, *ending-line***t***destination-line*

The **t** command duplicates the specified lines after the *destination-line*. Thus

```
1,$t$
```

duplicates the entire contents of the buffer that you are editing and places the duplication after the last line.

One use for **t** is to create a series of lines that differ only slightly. For example, 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]
t.                              [make second copy]
s/Now is/Yesterday was/         [change it]
```

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.
```

Moving Lines

The move (**m**) command enables you to move a group of lines from one place to another in the buffer. The **m** command syntax is

start-line,*end-line***m***destination-line*

When specifying the destination, remember that the text is moved to the line after the *destination-line*.

Suppose you want to move the first three lines of the buffer to the end. The following **move** command does it in a single step:

```
1,3m$
```

The lines to be moved can also be specified by context searches. Suppose the buffer contained the following text:

```
First paragraph
end of first paragraph.
Second paragraph
end of second paragraph.
```

You could reverse the two paragraphs by using the following command:

```
/Second/,/end of second/m/First/-1
```

Notice the -1. Because moved text is placed after the destination line, to place text before a line you must specify the destination line minus one. This is useful for moving text to the beginning of a file.

Dot is 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. If you are positioned on the first line, the command

```
m+
```

moves dot to one line after the current line. If you are on the second line, the command

```
m--
```

moves dot to one line before the current line.

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 have to take care to specify them properly, or you may not move the lines you want. Doing the job one step at a time makes it easier to verify at each step that you accomplished what you wanted. It is also a good idea to write out the buffer before doing anything complicated; then if you make a mistake, it's easy to back up to where you were.

Marking Your Spot in a File

The mark command (**k**) enables you to "mark" a line with a name and later reference the line by its mark name, regardless of its line number. This can be handy for moving lines and keeping track of them as they move. For example, typing

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

For example, find the first line of the block to be moved and mark it with

ka

Then find the last line and mark it with

kb

Go to the place where the text is to be inserted and type

'a,'bm.

Splitting Lines

ed provides a facility for splitting a single line into two or more shorter lines by "substituting in a carriage return." 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" by going to the line and typing:

```
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 carriage return 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 putting "very" on 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 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. Alone, **j** joins the lines signified by dot and dot + **1**. For example, suppose you have these two lines (with a space at the end of the first line)

```
Now is
the time
```

and you have just edited line 1, so dot is set to the first line. The command **j**, on a line by itself, joins the two lines together to produce

```
Now is the time
```

Any contiguous set of lines can be joined by specifying the starting and ending line numbers. For example

```
1,$jp
```

joins all the lines in a file into one big line and prints it.

Combining Files

The read command (**r**), like the edit command (**e**), enables you to read files into the editing buffer. However, there is a major difference between **r** and **e**; **r** appends the file to the buffer contents while **e** clears the buffer out before reading in the file. This factor makes **r** a useful tool for combining files. When the file is written out, it is written to the original file in the buffer.

One use of **r** would be composing a form letter. You could store a number of standard clauses or paragraphs in separate files and as you compose the letter, use **r** to read in the appropriate clauses and paragraphs and chain them together.

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. In **memo** 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, you must 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".

Editing Scripts

If a fairly complicated set of editing operations is to be done on a set of files, create a "script" (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.** (In the example, "..." represents additional **ed** command lines.) 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:

```
g/^@$/s//./g
```

Speeding Up Editing

One of the most effective ways to speed up editing is to enter commands without specifying line numbers. To do this, you must be able to determine what line a command will affect and what the current line will be after the command executes. 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 of the search.

The delete command, **d**, leaves dot pointing at the line that followed the last deleted line. When the line dollar ($) is 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. 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
.
```

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 filename
```

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 a range of line numbers, that range of lines is written out. The **w** command does not change dot; the current line remains the same, regardless of the lines written out.

(Since the **w** command is so easy to use, you should save what you are editing regularly just in case the system crashes, or in case you accidentally delete what you're editing.)

The general rule is simple; you are left on the last line changed. If there were no changes, then dot is unchanged. To illustrate, suppose that 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.

Summary of ed Commands

The 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 file name. 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 lines to the buffer after 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 that follows. The new lines are terminated by a period on a new line, as with **a**. If no lines are specified, replaces 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 deleted, so be sure to issue a **w** command first.
f	Prints the remembered file name. If a name follows **f**, then the remembered name is set to that name.
g	Global search--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 nonprinting characters and tabs visible. (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, prints the line specified by dot. A single line number is equivalent to a *line-number***p** command. A single RETURN prints ".+1", the next line.
q	Quits ed. Your work is not saved unless you first give a **w** command. Give **q** twice in a row to abort edit.
r	Reads a file into the buffer at the end unless specified elsewhere. Dot is set to the last line read.
s	Substitute--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.

u Undoes the last append, change, delete, insert, move, substitute (except global), or transfer command.

v Reverse of the **g** command, the command **v**/*string/commands* executes *commands* on those lines that do *not* contain *string.*

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*
Causes *command* to be executed as a XENIX command.

/string/
Context search. Searches for the next line containing *string* and prints the line. 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, and wraps around to **$.**

intel®

CHAPTER 5
vi: A VISUAL TEXT EDITOR

Introduction

vi (which stands for "visual") combines line-oriented and screen-oriented features into a powerful set of text editing operations that will satisfy any text editing need. Many of the commands used in **ed** are part of the **vi** command set because **vi** is a superset of **ed.** If you have already learned **ed,** you are well on your way to learning **vi.**

Begin by learning the features you will use most often. If you are an experienced user of **vi,** you may choose to refer to the **vi** entry in the *XENIX 286 Reference Manual*; more advanced features are covered there.

Demonstration

This section of the chapter walks you through creating a file when you invoke **vi,** entering text into the file, and exiting **vi.**

To enter **vi** and create a file named **temp,** type

```
vi temp
```

Your screen will look like this:

```
_
~
~
~
~
~
~
~
~
~
~
~
~
```

Note that only twelve lines are shown to save space. The tildes (~) will fill the editing window on your screen.

The top line of the display is where you will begin to enter text into the file; tildes indicate lines on the screen only, not actual lines in the file.

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

```
i
```

Like most **vi** commands, **i** doesn't appear on the screen. The command switches you from command mode to insert mode. Type in the following text:

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

If you make a mistake while typing, simply use the BACKSPACE key to erase the error. When you finish typing in this text, press the ESCAPE key. To exit **vi**, press and hold the SHIFT key and press "ZZ".

Basic Concepts

This section introduces the basic concepts of **vi** that you will need to use the editor effectively and efficiently. Here you will learn about the editing buffer, how to enter editor commands, the syntax of **vi** commands, and the importance of line numbers.

Entering vi

You can enter **vi** in several ways, depending on what you are planning to do. This section describes the different methods of entering **vi.**

Specifying a Single File

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 with the specified name is created.

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

vi +100 *filename*

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

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

vi +/*word filename*

The cursor is placed at the beginning of the line containing *word*. For example, to begin editing the file **temp** at the the first occurrence of "contain", type

```
vi +/contain temp
```

Specifying a Series of Files

If you have many files to edit in one session, you can invoke **vi** and specify more than one file name. In this way you can edit multiple files without leaving the editor to call each file. The command syntax for entering **vi** with multiple files is

```
vi file1 file2 file3 file4 file5 file6
```

Typing out many file names is tedious, and you may make a mistake. If you mistype a file name, you must either backspace and correct the typing error, or kill the whole line and retype it. A more convenient method is to invoke **vi** and use the metacharacters as abbreviations. To invoke **vi** on the above files without typing each name, type

```
vi file[0-9]
```

This invokes **vi** on all files that begin with the letters "file" followed by a single digit. You could also use the metacharaters **.** and *****, which would call any files beginning with "file" followed by a single character and any characters, respectively. You can plan your file names 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 file name 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].txt
```

When you invoke **vi** and specify more than one file name, you will see the following message when the first file is displayed on the screen:

x files to edit

where *x* is the number of files to be edited.

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. You need not specify a file name; the files are invoked in alphabetical order (or numerical, if the file names begin with numbers).

If you forget what files you specified when you invoked **vi**, type **:args** and **vi** displays the names of the files with the current file enclosed in brackets. You can determine the name of the file you are currently editing by typing **:file**, **:f**, or CONTROL-G.

To edit a file out of order, such as **file4** after **file2**, type

```
:e file4
```

instead of the **next** command. Type

```
:n
```

after you finish editing **file4** and you will go back to **file3**.

If you wish to start again from the beginning of the list, type

```
:rew
```

To discard all the changes you made and start again at the beginning, type

```
:rew!
```

Calling a File without Leaving vi

Occasionally you may find that changes you have made to one file affect another, so you need to edit the second file. Rather than exiting **vi** and re-entering, the **:e** command enables you to call the second file from within **vi**. The syntax of **:e** is

:e *filename*

Before calling the second file to the screen, be sure to save the original file because the editing buffer will be cleared before reading in the second file.

Exiting vi

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 RETURN. This command replaces the old copy of the file with the new one you have just edited, quits the editor, and returns you to the XENIX shell.

Similarly, if you type

```
ZZ
```

the same thing happens, except the old copy of the 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 made to the file, type

```
:q!
```

The exclamation point tells **vi** to quit unconditionally. If you leave out the exclamation point, as in

```
: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!** to leave the editor without saving your file.

Leaving vi Temporarily

While in **vi**, you may find it necessary to execute a XENIX command or a sequence of XENIX commands. Entering the command from **vi** and preceding it with an exclamation point (!) enables you to do this. The syntax for executing XENIX commands from **vi** is

:!*command*

For example, to determine the date, enter the command

```
:!date
```

XENIX responds with the current date and time and the message

```
[Hit return to continue]
```

Press RETURN to return to **vi** and the file you are editing.

To issue a series of XENIX commands, enter the command

```
:!sh
```

XENIX places you in the shell and displays the shell prompt (%) and the message

```
[Hit return to continue]
```

Issue any XENIX commands necessary and then enter CONTROL-D to exit the shell; the message

```
[Hit return to continue]
```

is again displayed. Press RETURN to get back to **vi** and the file you are editing.

It is a good idea to save your file before executing any XENIX commands, just in case something goes wrong.

Line Numbers

As stated earlier, **vi** is a combination line-oriented and screen-oriented text editor. Many editing commands will take either single line numbers or line number ranges as prefixing arguments. These arguments normally specify the lines in the editing buffer to be affected by the given command. Any time a command changes the number of lines in the editing buffer, **vi** renumbers the lines. At all times, every line in the editing buffer has a line number. In **vi**, the current line, word, or character is the line, word, or character where the cursor is positioned. You can determine the actual line number of the current line by entering

```
:nu
```

vi will print the current line number and then display the line.

vi also enables you to turn on a function that displays the line number next to each line. To turn on the **number** function, type

```
:set number
```

To turn off this function, type

```
:set nonumber
```

The Editing Buffer

Each time you invoke **vi**, an area in the memory of the computer is allocated where you will perform all editing operations. This area is called the "editing buffer." When you edit a file, a copy of the original is placed in the editing buffer, where you will work on it. Only when you write out the buffer do you change the original file.

Writing Out the Editing Buffer

You will probably want to save your text for later use. To write out the contents of the buffer into a file, use the **write (:w)** command followed by the name of the file you want to write to. This copies the contents of the buffer to the specified file, destroying any previous contents of the file. For example, to save the buffer in a file named **text**, type

```
:w text
```

Leave a space between **:w** and the file name. **vi** responds by displaying the file name, number of lines, and number of characters. (Blanks and the RETURN character at the end of each line are included in the character count.) Writing out a file makes a copy of the text in the editing buffer--the buffer's contents are not disturbed, so you can continue editing it. When you invoke **vi** by using a *filename* argument, a **:w** command by itself writes the buffer out to *filename*.

Writing out the editing buffer occasionally as you work is an excellent safety measure; if the system crashes all the text in the buffer is lost, but any text written out is safe.

Commands

Enter commands by typing them at the keyboard and then pressing RETURN. Most commands are single characters, although some require a preceding colon to signify a special operation such as writing out the buffer. Unless otherwise specified, **vi** commands operate on the line where the cursor is positioned; this is called the "current line." Many commands take file names or string arguments that are used by the command when it is executed.

Repeating Commands

The **repeat** command (**.**) enables you to repeat the last executed screen-oriented **vi** command. 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.

Undoing Commands

Any editing command can be negated with the **undo** (**u**) command. If you have deleted a line and then decide you wish to keep it, press **u** and the line reappears. Use the following line as an example:

```
Text contains lines.
```

Place the cursor over the c in "contains", then delete the word with the **dw** command. Your screen should look like this:

```
Text lines.
```

Press **u** to undo the **dw** command. The word "contains" reappears:

```
Text contains lines.
```

If you press **u** again, "contains" is deleted again:

```
Text lines.
```

Note that **u** negates only the previous command. For example, if you make a global search and replace, then delete a few characters, pressing **u** will undo the last character deletion but not the global search and replace.

Performing a Series of Line-Oriented Commands

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 **vi.** 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 **vi,** type

```
vi
```

Moving in a File

vi provides two methods of moving around in file: moving the cursor and scrolling. The following sections describe these methods.

Moving the Cursor

When editing a file, the cursor shows your position in the file. You can move the cursor in two ways: use the keyboard direction arrows or enter commands that tell the cursor where to move to. The direction arrows move the cursor one character at a time. The cursor control commands enable you to move the cursor to a specific location. The cursor control commands are

h	Moves cursor 1 space left
l	Moves cursor 1 space right
SPACE BAR	Moves cursor 1 space right
b	Moves cursor 1 word left
w	Moves cursor 1 word right
k	Moves cursor 1 line up
j	Moves cursor 1 line down
RETURN	Moves cursor 1 line down
)	Moves cursor to end of sentence
(	Moves cursor to beginning of sentence
}	Moves cursor to beginning of paragraph
{	Moves cursor to end of paragraph
CONTROL-W	Moves cursor to first character of insertion

Scrolling

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

CONTROL-U	Scrolls up one-half screen
CONTROL-B	Scrolls up a full screen
CONTROL-D	Scrolls down one-half screen
CONTROL-F	Scrolls down a full screen
z + RETURN	Scrolls the current line to the top of the screen

To place a specific line at the top of the screen, precede the "z" with the line number, as in

```
33z
```

Press RETURN, and line 33 scrolls to the top of the screen.

Inserting Text

vi has six commands that enable you to insert text into a file. Any time you place text in a file, whether it is a new, empty file or an existing file, you will use one of these six commands:

a	Begins text insertion after the cursor
A	Begins text insertion after the last character on a line
i	Begins text insertion before the cursor
I	Begins text insertion before the first character on a line
o	Begins text insertion on next line down
O	Begins text insertion on the line above

All insert commands are terminated by pressing the ESCAPE key.

While most of the commands are self-explanatory, the **o** and **O** commands can be confusing and require more explanation. **o** and **O** are line editing commands and are used to insert lines of text rather than just a word or character. Regardless of where the cursor is on the current line when you enter one of the "o" commands, a blank line is inserted between the current line and either the line above or below. When the blank line appears on the screen, begin entering the new text. When you are finished entering text, press ESCAPE. At the end of the text, do not press RETURN unless you want a blank line to appear in the text.

Occasionally while you are editing a file, you may find that you need to repeat some text frequently. Typing the same text repeatedly is tedious, time-consuming, and error-prone. **vi** has a feature that enables you to repeat the last insertion, to a maximum of 128 characters. To repeat the last text insertion, follow these steps:

1. Position the cursor at the next location where the text is to be inserted.
2. Type **i** to enter insert mode.
3. Enter CONTROL-@.

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 CONTROL-V before typing the control character. CONTROL-J, CONTROL-Q, and CONTROL-S cannot be inserted as text. (CONTROL-J is a carriage return and CONTROL-Q and CONTROL-S are meaningful to the operating system and are trapped before they are interpreted by **vi**.)

Deleting Text

You can delete characters, words, or lines from a file by using the following delete commands:

x	Deletes a single character at the cursor position
*n***x**	Deletes forward from the current cursor position, *n* number of characters
X	Deletes the character immediately preceding the current cursor position
*n***X**	Deletes backward from the current cursor position, *n* number of characters
dw	Deletes the single word where the cursor is positioned
d0	Deletes all text from the cursor position to the beginning of the current line
d$	Deletes all text from the cursor position to the end of the current line
D	Deletes all text from the cursor position to the end of the current line
*n***dw**	Deletes forward from the current cursor position, *n* number of words
dd	Deletes the current line
*n***dd**	Deletes forward from the current cursor position, *n* number of lines

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

```
3x
```

To delete three characters preceding the cursor, type

```
3X
```

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. To delete three words, type

```
3dw
```

The **d$** and **D** commands delete all text following the cursor on the current line, including the character the cursor is resting on. The **dd** command deletes a specified number of lines and closes up the space. If no number is given, only the current line is deleted. To delete three lines, type

```
3dd
```

Another way to delete several lines is to use a line-oriented command. To use this command, you need to know the line numbers of the text you wish to delete.

To delete lines 200 through 250, type

```
:200,250d
```

Press RETURN. When the command finishes, the message

```
50 lines
```

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

You can remove lines without displaying line numbers by using shorthand addresses. For example, to remove all lines from the current line (the line the cursor rests on) to the end of the file, 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

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

To delete the current line and 3 lines preceding it, type

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

For more information on using addresses in line-oriented commands, see **vi** in the *XENIX 286 Reference Manual.*

If you wish to delete all of the text you just typed, press CONTROL-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 ESCAPE, any text remaining from the original insertion disappears.

Copying Text

Commands in **vi** enable you to copy text from other files and from the file you are editing and place it in a designated loaction within the current file. The next two sections describe these procedures.

Copying Text from Other Files

To insert the contents of another file into the file you are currently editing, use the **read** command. Move the cursor to the line immediately *above* the place you want the new material to appear, then type

:r *filename*

where *filename* is the file containing the material to be inserted. 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.

A derivative of the **:r** command enables you to execute a XENIX command and place the result into the editing buffer at a designated location. Position the cursor anywhere on the line *above* the position where you want the result placed and type

```
:r!command
```

Inserting lines from another file is more complicated. The selected lines are copied from the source file into a buffer and then inserted into the destination file. To select the lines to be copied, save your original file with the **write** command (**:w**), but do not exit **vi.**

Type

```
:e filename
```

where *filename* is the file that contains the text you want to copy.

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

Type

```
mk
```

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

Move the cursor to the last line of the selected text and type

```
"ay'k
```

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

Type

```
:e#
```

to return to your destination file. Move the cursor to the line *above* the place you want the new text to appear, then type

```
"ap
```

This inserts a copy of the lines in buffer **a** into the destination file and places the cursor on the first letter of this new text. Buffer **a** still contains the original lines.

You may have a maximum of 26 buffers, named with the single lowercase letters a-z. To name and select different buffers, replace the **a** in the above examples with whatever letter you wish.

Copying Text from Elsewhere in the File

To copy text from one place in a file to another place in the same file, use the copy (**co**) command. **co** is a line-oriented command, and to use it you must know the line numbers of the text to be copied and the destination. The "Basic Concepts" section of this chapter describes how to determine line numbers in **vi**. The syntax of the **co** command is

:*beginning-line, ending-line* **co** *destination-line*

where *beginning-line* is the line number of the first line of text you want copied, *ending-line* is the line number of the last line of text you want copied, and *destination-line* is the line number of the line of text that *precedes* the location where you want the text copied to. You may specify either a single line number or a range of line numbers for the beginning and ending line numbers. If you specify a range of line numbers, the numbers must be separated by a comma.

For example, to copy lines 10, 11, 12, 13, and 14 of a file to a position between lines 30 and 31, use the following command

```
:10, 14 co 30
```

If you have text that is to be inserted 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

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

move the cursor over the "F" in "Files". Type the following line, which will not be echoed on your screen:

```
"ayy
```

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

Type the following line:

```
"ap
```

This inserts a copy of the line in buffer **a** into the file and places the cursor on the first letter of the new text. The buffer still contains the original copy of the line.

Your screen looks like this:

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

If you wish to copy several consecutive lines, indicate the number of lines you wish to copy after the name of the buffer. For example, to place three lines of a file into the buffer **a**, type

```
"a3yy
```

Moving Text

To move a block of text from one place in a file to another, use the move (**m**) command. The **m** command is a line-oriented command, which means you must know the line numbers of file to use this command. The syntax of the **m** command is

> **:***beginning-line, ending-line* **m** *destination-line*

For example, to move lines 10, 11, 12, 13, and 14 of a file to a position between lines 30 and 31, use the following command

```
:10, 14 m 30
```

vi also enables you to remove text from a file, store the text in a special buffer called a "delete buffer," and reinsert the text at any number of locations in the file. There are a total of nine delete buffers. The nine most recent text deletions are stored in the delete buffers, with the most recent being stored in buffer 1 and the least recent stored in buffer 9.

In other words, the first deletion in an editing session goes into buffer 1. The second deletion goes into buffer 1 and pushes the original contents of 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; and so on until all nine buffers are used. When a tenth deletion is made, the deleted text is placed in buffer 1, the contents of the remaining buffers are shifted, and the original contents of buffer 9 are deleted.

Text remains in the delete buffers until it is pushed off the stack or until you exit the editor, making it possible to delete text from one file, change files without leaving the editor, and place the deleted text in another file.

Using the following text as an example

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

delete the first line by typing

```
dd
```

Move the cursor to the second line and delete it, then move the cursor to the last line of text, and type

```
"1p
```

The line from the *second* deletion appears:

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

Now type

```
"2p
```

The line from the first deletion appears:

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

Inserting text from a delete buffer does not remove the text from the buffer. Since the text remains in a buffer until it is either pushed off the stack or until you quit the editor, you may use it as many times as you wish. You can also place text in named buffers.

Joining and Breaking Lines

To join two lines press

```
J
```

while the cursor is on the first of the two lines you wish to join. Notice that this is an uppercase J.

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

```
r
```

then press RETURN.

Searching

You can search forward and backward for patterns in **vi.** The syntax for searching forward is

/pattern

and the syntax for searching backward is

?pattern

If the specified pattern exists, the cursor moves to the first character of the pattern. For example, to search forward in the file for the word "account", type

```
/account
```

The cursor is placed on the first character of the pattern. To place the cursor at the beginning of the line above "account", for example, type

```
/account/-
```

To place the cursor at the beginning of the line two lines above the line that contains "account", type

```
/account/-2
```

To place the cursor two lines below "account", type

```
/account/+2
```

To search backward through a file, use **?** instead of **/** to start the search. For example, to find all occurrences of "account" above the cursor, type

```
?account
```

To search for a pattern containing any of the metacharacters (. * [ ] $ \ and ^), each metacharacter must be preceded by a backslash. For example, to find the pattern "U.S.A.", type

```
/U\.S\.A\./
```

You can continue to search for a pattern by pressing

```
n
```

The pattern is unaffected by intervening **vi** commands, and you can use **n** to search for the pattern until you type in a new pattern or quit the editor.

vi searches for exactly what you type. For instance, if the pattern you are searching for begins with an uppercase letter, **vi** ignores all occurrences of the pattern that begin with a lowercase letter.

By default, searches "wrap around" the file. That is, if a search starts in the middle of a file, when **vi** reaches the end of the file it will "wrap around" to the beginning and continue until it returns to where the search began. Searches will be completed faster if you specify forward or backward searches, depending on where you think the pattern is.

Searching and Replacing

The search and replace commands enable you to perform complex changes to a file in a single command. Learning how to use these commands is a must for the serious **vi** user.

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 following file:

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

To replace the word "contain" with the word "are" throughout the file, type the following command:

```
:g/contain /s//are /g
```

This command says "find every occurrence of the pattern 'contain' on each line in the file and substitute for that pattern the word 'are' everywhere it occurs." Note that a space is included in the search pattern for "contain"; without the space "contains" would become "ares".

After the command executes, the screen should look like this:

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

To replace "contain" with "are" throughout the file and print every line changed, use the **p** command:

```
:g/contain /s//are /gp
```

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 CONTROL-L.

Sometimes you may not want to replace every instance of a given pattern. The **c** option displays every occurrence of *pattern* and waits for you to confirm that you want to make the substitution. If you press **y** the substitution takes place; if you press RETURN the next instance of *pattern* is displayed.

To run this command on the example file, type

```
:g/contain/s//are/gc
```

vi responds with the following display:

```
Text contains lines.
     ^^^^
```

Press "y", then RETURN. The line with the next occurrence of "contain" appears. **vi** always displays the whole line, but the pattern being searched for is underscored with carets. If you do not want the pattern to be replaced, enter "n" or RETURN.

Substituting Text

When using **vi,** you may substitute by using the following commands:

r	Begins text insertion on current character, replaces one character only
R	Begins text insertion on current character, replaces until ESCAPE
nsstring	Replaces *n* characters, from the cursor forward, with *string*
*n*S*text*	Replaces *n* lines, from the cursor forward, with *text*
cw	Changes 1 word
*n*cw	Changes *n* words
C	Replaces text from the cursor to the end of the line
cc	Changes current line
*n*cc	Changes *n* lines

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

```
rb
```

If a number is given before **r,** that number of characters is replaced with the next character typed. For example, to replace the character above the cursor, plus the next three characters, with the letter "b", type

```
4rb
```

Note that you now have four "b's" in a row.

The **s** command replaces a specified number of characters, beginning with the character under the cursor, 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 given, 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.

The **cw** command replaces a word with text you type in. For example, to replace the word "bear" with the word "fox", move the cursor over the "b" in "bear". Enter

```
cw
```

A dollar sign appears after the "r" in "bear", marking the end of the text that is being replaced. Type

```
fox
```

The rest of "bear" disappears and only "fox" remains.

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 ESCAPE. The result is

```
Who's afraid of the little lamb?
```

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.

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 sentence

```
In vi, removing a line is as easy as removing a letter.
```

move the cursor to the beginning of the line and type

```
:s/removing/deleting/g
```

This line-oriented command means "substitute (s) the word 'deleting' for the word 'removing', everywhere 'removing' occurs on the current line (g)". If you don't include a **g** at the end of the command line, only the first occurrence of "removing" is changed.

Metacharacters

Metacharacters are a set of special characters used to describe text patterns in search and substitute commands. These patterns are called "regular expressions" and occur in several other important XENIX commands and utilities, including **grep** and **sed** (see the *XENIX 286 Reference Manual*). A complete list of the metacharacters follows:

```
\    .    ^    $    *    []    &
```

The following sections describe how to use these metacharacters in search and substitute commands.

Backslash

The backslash (\) turns off any special meaning that the following character has. As an example, the sequence

```
\.
```

in a search or substitute command changes the meaning of the period from "match any single character" to a literal period. When you are adding text with **a**, **i**, or **c**, the backslash has no special meaning.

Period

When used in a search or on the left-side expression of a substitute command, the period stands for any single character (this is frequently called a "wildcard" character). For example, the search command

```
/x.y/
```

finds any line where "x" and "y" occur separated by any single character, as in

```
x+y
x-y
x y
xzy
```

If you use a period in the right-side expression of a substitute command, the period assumes its literal meaning.

Caret

The caret (^) signifies the beginning of a line; the first character of every line is a caret (although it is invisible). To search for or substitute an expression at the beginning of a line, precede the expression with a caret. The search and substitute commands have the following formats when using the caret:

/^*expression*/

s/^*expression*/*expression*/

For example, suppose you are looking for a line that begins with "the". If you simply type

```
/the/
```

you will probably find several lines containing "the" before arriving at the one you want. But with

```
/^the/
```

you narrow the search context to "the" at the beginning of a line only, and thus arrive at the desired line more quickly.

The caret (^) also enables you to insert characters at the beginning of a line. For example

```
s/^/ /
```

places a space at the beginning of the current line.

Dollar Sign

The dollar sign ($) signifies the end of a line; the last character of every line is a dollar sign (although it is invisible). To search for or substitute an expression at the end of a line, follow the expression with a dollar sign. The search and substitute commands have the following formats when using the dollar sign:

/*expression*$/

s/*expression*$/*expression*/

For example, suppose you are looking for a line that ends with "the". If you simply type

```
/the/
```

you will probably find several lines containing "the" before arriving at the one you want. But with

```
/the$/
```

you narrow the search context to "the" at the end of a line only, and thus arrive at the desired line more quickly.

The dollar sign ($) also enables you to insert characters at the end of a line. For example

```
s/$/./
```

places a period at the end of the current line.

Star

The star is useful for finding numerous occurrences of a single character. Literally, it means "find any number of consecutive occurrences, including zero, of the character that preceded the star followed by any designated text." For example, the command

```
/n*o
```

finds any number of occurrences of the letter "n" (including zero occurrences) followed by the letter "o." When using the star in substitution command lines, be very careful; if you specify the star with the wrong context, the file may be incorrectly changed.

Suppose that the line you want to edit is

```
textx..................ytext
```

If you type

```
s/x.*y/x   y/
```

the result is unpredictable. If no other x's or y's occur on the line, the substitution will work, but not necessarily. The period matches any single character, so ".*" matches as many single characters as possible, and unless you are careful, it can remove more of the line than you expected.

Brackets

Brackets are used in search and substitution commands to specify a "character class". 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".

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 may appear within a character class, and only three special characters (^,], and -) appear inside the brackets; even the backslash doesn't have a special meaning. To search for any special characters, for example, type

```
/[.\$^[]/
```

Digits can be abbreviated as [0-9]; similarly, [a-z] stands for the lowercase letters, and [A-Z] for uppercase.

Ampersand

The ampersand (&) can be used on the right side of a substitute command to signify the string of text found on the left side of a substitute command. Using the ampersand eliminates repeating the string on both sides of the command and is extremely useful when adding text within a line. The syntax of the ampersand command is

s/*string*/& *new string*/

For instance, if you are editing the line

```
Now is the times
```

and want to change it to read

```
Now is the best of times
```

you would enter the command

```
s/the/& best of/
```

Here, the ampersand stands for "the", so that "the" is changed to "the best of".

The ampersand may be used more than once within the same command line; it always signifies the string on the left side of the substitution.

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 ESCAPE until the bell rings. When the bell rings you are in command mode.

- **I can't get out of a subshell.**

  Press CONTROL-D to exit any subshell. If you have created more than one subshell (not a good idea, usually), keep pressing CONTROL-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 CONTROL-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, type

  ```
  stty sane
  ```

  then press CONTROL-J or LINEFEED. Pressing CONTROL-J instead of RETURN is important here, since the RETURN key may 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 terminal 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, the file is irretrievably lost.

- **I keep getting a colon on the status line when I press RETURN.**

  You are in line-oriented command mode. Return to normal **vi** command mode by typing

  ```
  vi
  ```

- **I get the error message "Unknown terminal type [Using open mode]" when I invoke vi.**

  Your terminal type is not set correctly. Press ESCAPE, then type

  ```
  :wq
  ```

  and press RETURN.

Summary of vi Commands

vi *filename*	Enters **vi** and starts edit of *filename* at line 1
vi +*n filename*	Enters **vi** and starts edit of *filename* at line *n*
vi + *filename*	Enters **vi** and starts edit of *filename* at last line
vi +/*pattern filename*	Enters **vi** and starts edit of *filename* at *pattern*
vi -r *filename*	Recovers *filename* after a system crash
h	Moves cursor 1 space left
l	Moves cursor 1 space right
SPACE BAR	Moves cursor 1 space right
b	Moves cursor 1 word left
w	Moves cursor 1 word right
k	Moves cursor 1 line up
j	Moves cursor 1 line down
RETURN	Moves cursor 1 line down
)	Moves cursor to end of sentence
(	Moves cursor to beginning of sentence
{	Moves cursor to beginning of paragraph
}	Moves cursor to end of paragraph
CONTROL-W	Moves cursor to first character of insertion
CONTROL-U	Scrolls up one-half screen
CONTROL-D	Scrolls down one-half screen
CONTROL-F	Scrolls down one screen
CONTROL-B	Scrolls up one screen
i	Begins text insertion before the cursor
I	Begins text insertion before first character on the line
a	Begins text insertion after the cursor

A	Begins text insertion after last character on the line
o	Begins text insertion on next line down
O	Begins text insertion on the line above
r	Begins text insertion on current character, replaces one character only
R	Begins text insertion on current character, replaces until ESCAPE
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
cw	Changes 1 word
*n*cw	Changes *n* words
cc	Changes current line
*n*cc	Changes *n* lines
/*pattern*	Finds the next occurrence of *pattern*, including the pattern found within other words
?*pattern*	Finds the previous occurrence of *pattern*, including the pattern found within other words
/^*pattern*	Finds next line that starts with *pattern*, including words beginning with that pattern
/[tT]ext/	Finds the next occurrence of "text" or "Text"
n	Repeats the most recent search, in the same direction
:s/x/y/g	All occurrences of "x" become "y" 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)

*	Matches any number of consecutive occurrences (including 0) of the character preceding it
^	Matches beginning of line special character
$	Matches end-of-line special character
.	Matches any single character
[ ]	Matches a range of characters
: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)
:nu	Displays the current line
:file	Displays the name of the current file
:f	Displays the name of the current file

intel®

CHAPTER 6 mail: THE XENIX MAIL SYSTEM

Introduction

The XENIX mail system is a versatile communication facility that enables 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 the 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.

Basic Concepts

This section introduces the basic concepts of **mail** that you will need to use **mail** effectively and efficiently. Here you will learn about getting help, using mailboxes, writing messages, entering and exiting **mail**, and performing simple functions.

Mailboxes

Think of the **mail** system as a small-scale postal system. The "post office" is called the "system mailbox" in XENIX. The system mailbox contains a file for each user in the directory **/usr/spool/mail.** Your personal mailbox, or "user mailbox," is the file **mbox** located in your home directory. Mail sent to you is held in the system mailbox until the next time you log in; when you log in, the system notifies you that you have mail. After reading your mail, it is saved in your user mailbox (**mbox**) if you do not delete it. Note that the user mailbox differs from a real mailbox in several respects:

- The user mailbox is *not* the place where mail is initially routed--that place is the system mailbox in the directory **/usr/spool/mail.**
- Mail is not picked up from your user mailbox; it is picked up from the system mailbox.

Modes of Operation

XENIX **mail** has three modes of operation: reading mail (command mode), sending mail (compose mode), and editing messages (edit mode).

Command mode is the mode you enter when there is mail waiting to be read. It is accessed by using the **mail** command with no user name. From command mode you read mail, execute shell commands, change **mail** options, and send mail to other users. You can access both compose mode and edit mode from command mode.

If you invoke **mail** with a user name, you are placed in the **mail** compose mode when **mail** comes up. Compose mode enables you to write messages. Special commands within compose mode enable you to save portions of messages being composed, display messages being composed, and even read in other files. You can access edit mode from compose mode.

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 one of the text editors. To enter edit mode from command mode, use either the **e (edit)** command to enter **ed**, or the **v (visual)** command to enter **vi.** To enter edit mode from compose mode, use the commands **~e (ed)** or **~v (vi).**

Getting Help

The **help (?)** command displays a brief summary of all **mail** commands, so if you ever get stuck while in command mode, type

```
?
```

or

```
help
```

To get help while you are in compose mode, use **~?.** Edit mode has no **help** facility.

Message Format

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

To:	This field is mandatory and contains the names of the users you're sending mail to.
Subject:	This optional field contains text describing the message.
Cc:	The optional carbon copy field contains the names of those users who will receive copies of a message. Message recipients see these names in the received message.
Bcc:	The optional blind carbon copy field contains the names of users who will receive copies of a message. Recipients do *not* see these names in the received messages.
Return-receipt-to:	This optional field contains the names of those users who will receive an automatic acknowledgement of the message.

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

Entering and Exiting mail

You can enter **mail** in two ways. If you have mail waiting to be read, just type

```
mail
```

and press RETURN to enter the **mail** system in command mode and read your messages. If you don't have any mail waiting, the system displays the message

```
No messages.
```

and returns to the shell.

The second way to enter **mail** is to use the syntax

mail *username*

where *username* is the name of a user that you want to send mail to (this can even be yourself). This method gets you into the **mail** system in compose mode.

To leave **mail** from command mode, you have a choice of three commands: **x** (for exit), **q** (for quit), and CONTROL-D, which functions the same as **q**.

When you exit **mail** by using **x**, the system returns to the shell without making any changes to the system or user mailbox; no messages are saved or deleted. The next time you log in, you will have the same messages waiting as well as any new messages that have been received.

Exiting **mail** with **q** or CONTROL-D will alter the system and user mailboxes. Deleted messages are discarded and saved messages are stored in their designated files.

To leave **mail** from compose mode, use CONTROL-D at the start of a new line. The message is sent to the user(s) specified when you entered **mail.**

Message Headers

When you enter **mail** without a user name and there is mail waiting, a list of numbered message headers is displayed. A header is a single line of text containing descriptive information about a message. (Note that the word "heading" describes the first part of a message, and "header" describes **mail's** one-line description of a message.) The header information includes

- The message number
- The sender's name
- The date sent
- The number of lines and characters
- The subject (if the message contains a **Subject:** field)

Message headers are displayed in "windows" with the **headers** command. A header window contains no more than 18 headers. If fewer than 18 messages are in the mailbox, all are displayed in one header window. If there are more than 18 messages, then the list is divided into an appropriate number of windows. You can move forward and backward one window at a time with the **headers** + and **headers** - commands.

Command Syntax

Each **mail** command has its own syntax. Some commands take no arguments, some take only one, and others take several arguments. The more flexible commands, such as **print**, accept combinations of lists of message numbers (message-lists) and user names. For these commands, **mail** first gathers all message numbers and ranges from the message headers, 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 numbered from 4 to 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 "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, any arguments you give are ignored and no error occurs. If no message-list is specified for a command that requires a message-list argument, the **mail** command uses the number of the last message displayed as the message-list argument. If that message 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 backward, and if there are no good messages at all, **mail** types

```
No applicable messages
```

Specifying Messages

Commands such as **print** and **delete** can be given a message-list argument to apply to several messages at once. Thus **delete 2 3** deletes messages 2 and 3, while **delete 1-5** deletes messages 1 through 5. A star (*) addresses all messages, and a dollar sign ($) addresses the most recent (highest numbered) message. The **top (t)** command prints the first five lines of a message; hence, you can type

```
top *
```

to print the first five lines of every message. Message-lists can contain combinations of lists, ranges, and names. For example, the following command prints out all messages from Tom or Bob and with the header numbers 2, 4, 10, 11, or 12:

```
p tom bob 2 4 10-12
```

Executing Shell Commands

To execute a shell command from command mode, precede the command with an exclamation point. For example

```
!date
```

prints out the current date without leaving **mail.** You can also execute shell commands from compose mode with the syntax

~!*command*

Determining the Number of the Current Message

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

Counting the Number of Characters in a Message

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 Working Directories

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

Reading Commands from a File

The **source (so)** command reads in **mail** commands from the specified file. This enables you to write a file to perform routine **mail** functions, run the file while you are away from your work area, and review your messages at any time. The syntax for the **so** command is

so *filename*

Reading mail

To read messages sent to you, type

```
mail
```

mail then brings your mail from the system mailbox and prints out a one-line header for each message. The most recent message is the first message listed and may be printed by entering

```
p
```

You can move forward one message by pressing RETURN or by typing "+". To move forward *n* messages, use *+n*. You can move backward one message with the - command or move backward *n* messages with *-n*. You can also move to any message and display 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 are displayed.

Suppose 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  1:23 6/84 "Invite"
```

and you want to read the second message, type

```
2
```

and press RETURN. This causes **mail** to respond with

```
From sam Tue Sep 20 22:55 1983
To: john
Subject: Meeting

(text of message)
```

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

```
.
```

prints the current message. The command

```
^
```

prints message 1. The command

```
$
```

prints message 3.

Displaying the First Five Lines

The **top** (**t**) command prints the first five lines of each specified message. Message numbers may be specified individually, collectively, or as a range. For example

```
top 2-12
```

prints out the first five lines of each of the messages 2 through 12. In the example

```
top 2,4,6,8
```

the first five lines of messages 2, 4, 6, and 8 are displayed.

Editing a Message

To edit individual messages using the text editor **ed**, use the **edit** (**e**) command. It takes a message-list and processes each message in turn by writing it to a temporary buffer. 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. The mail system then 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.

Displaying the Next Message

The **autoprint** switch (see description automatically prints the next message in the list when you delete the current message. Also, the restored message is displayed automatically after execution of an **undelete** command.

Listing Messages in Chronological Order

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

The **mchron** switch also prints messages in chronological order but lists them in the opposite order, i.e., highest-numbered (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.

Replying to mail

Often you want to deal with a message by responding to its author right away. The **reply (r)** command is useful for this purpose: it takes a message-list and sends mail to the author of each message. The original message's subject field is copied as the reply's subject. Each message is composed in compose mode and messages are terminated by pressing CONTROL-D.

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

Saving mail

The save (s) command enables you to save messages in files other than **mbox.** By using **save,** you can organize your mail by putting messages in appropriate files. The **save** command writes out each message to the file given as the last argument on the command line. For example, the following command appends messages 1-5 to the file **letters:**

```
s 1-5 letters
```

The file **letters** is created if it does not already exist. Each saved message is marked with a star (*).

save writes out the entire message, including the **To:, Subject:,** and **Cc:** fields. In comparison, the **write** command (discussed below) writes out only the bodies of the specified messages.

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

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

writes out the bodies of all messages from John and Elliot in the number range 3-17. They are catenated to the end of the file named **book.**

The **mbox (mb)** command marks each message specified in a message-list so that they 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.**

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.

Deleting Messages

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

```
delete 1
```

prevents **mail** from retaining message 1 in the system mailbox. The message and its number disappear; the remaining messages in the queue are not renumbered.

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.

Undeleting Messages

The **undelete (u)** command causes a message previously deleted with **d** or **dp** to reappear. For example, to undelete message 3, type

```
u3
```

You cannot undelete messages from previous **mail** sessions; they are unrecoverable.

Forwarding mail

To forward a copy of the current message to another user, use the forward **(f)** command. The syntax of the **f** command is

f *usernames*

For example, to forward the current message to 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 works the same as the **f** command, except that the forwarded message is not indented.

Printing mail

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

```
l doug
```

prints out each message from Doug.

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 CONTROL-D on a blank line. The message is automatically sent to the specified people. The section "Composing Messages" that follows describes some **mail** features that help you compose messages.

If you have a file named **letter** that contains a written message, you can send it to Sam, Bob, and John by typing

```
mail sam bob john <letter
```

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.

To send mail from command mode, 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. Note that the **mail** command is in most ways identical to typing **mail** *usernames* at the XENIX command level.

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** in your home directory, or be notified via return mail, in which case a copy is included in the return mail message.

Composing Messages

To compose messages, you must enter **mail** 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 when entering text, including CONTROL-U to kill a line and BACKSPACE to back up one character. Note that entering two interrupts in a row (i.e., pressing INTERRUPT twice) aborts your composition.

In compose mode, **mail** treats lines beginning with the tilde (~) in a special way. This character introduces compose mode commands called compose escapes; compose escapes enable you to enter commands and perform functions while composing messages.

For example, typing

```
~m
```

by itself on a line places a copy of the most recently printed message inside the message you are composing. The copy is shifted right one tab stop. Other compose escapes set up heading fields, add and delete recipients to the message, enable you to escape to an editor, enable you to revise the message body, or run XENIX commands. For a list of the available compose escapes when in compose mode, type

```
~?
```

The following sections contain more detail about the compose escapes.

Displaying Messages

To display the text of a message you are composing, type

```
~p
```

This prints a line of dashes and the heading and body of the message so far.

Editing Messages

If you are dissatisfied with a message as it stands, you can edit the message by invoking **ed** with the editor escape, **~e.** This causes the message to be copied into a temporary buffer so that you can edit it. Similarly, the **~v** escape causes the message to be copied into a temporary buffer so that you can edit it with **vi.** After modifying the message to your satisfaction, write it out and quit the editor. **mail** responds by displaying

```
(continue)
```

after which you may continue composing your message.

Editing Headers

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

```
~t name1 name2 ...
```

naming 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, 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 look at the message by typing **~p**, which prints out all heading fields and the body of the message.

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.

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 their names do not appear in the header or message.

The escape

```
~R
```

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

If you wish to edit any header fields in ways not possible with the **~t**, **~s**, **~c**, and **~R** escapes, you can use

```
~h
```

The escape **~h** prints **To:** followed by the current list of recipients and leaves the cursor at the end of the line. If you enter ordinary characters, they are appended to the end of the current list of recipients. You can also use the normal XENIX command line editing characters to edit these fields, so you can erase existing heading text by backspacing over it. Press RETURN to advance through the fields. Each of these fields can be edited in the same way. A final RETURN appends text to the end of the message body. You can use **~p** to print the current text of the heading fields along with the body of the message.

Adding a File to a Message

You may want to include the contents of some file in your message. The escape

```
~r filename
```

appends the named file to your current message. The system returns an error 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** from 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

If you are sending mail from within **mail**'s command mode, 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 Messages in a File

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

```
~w filename
```

The mail system 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.

Escaping to the Shell

To temporarily escape to the shell, use the escape

```
~!command
```

This executes *command* and returns you to **mail** compose mode without altering your message. 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. This escape is particularly useful with the **fmt** command that performs simple formatting operations on the text of your message. If the command produces no output, **mail** assumes that something is wrong, retains the old version, 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. Note that this escape does not work if you enter compose mode from the XENIX shell. You will receive the message

```
May not execute cmd while composing
```

Placing an Escape Character at the Beginning of a Line

If you wish to send a message that contains a line beginning with an escape character (e.g., tilde ~), you must type the escape character twice. For example, typing

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

appends

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

to your message.

Sending Network mail

Mail can be sent between XENIX machines connected with Micnet by specifying a machine name and the user name on that machine, separated by a colon:

machine:user

If appropriate gateways are known to the system, you can send mail to sites within the **uucp** network using the syntax

machine!user

(Be sure to escape the exclamation point by preceding it with a backslash (\) when giving it on a **csh** command line.) **mail** may also interpret other characters in the mail path when dealing with other networks. In most cases, aliases should be set up so that specifying machine names is unnecessary. For more information about sending network mail, see the *XENIX 286 Communications Guide.*

Setting Up Your mail 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. Systemwide aliases are defined in **/usr/lib/mail/mailrc.** Personal aliases and **set** options are defined in **.mailrc.** The following is a sample **.mailrc** file:

```
# pound sign introduces comments
# personal aliases office and cohorts are defined below
alias office bill steve karen
alias cohorts john mary bob beth mike
# set dot lets messages be terminated by period on new line
# set askcc says to prompt for Cc: list after composing message
set dot askcc
# cd changes directory to different current directory
cd
```

Setting Options

mail has several options that can be specified from **mail** command mode or in the file **.mailrc** in your home directory.

When you use the **mail** command mode to set and unset **mail** options, the settings that you specify during a session are in effect only during that session. If you want to change options permanently, you must edit the **.mailrc** file.

The command

```
set ?
```

prints out a list of the available options.

mail switch and string options are set with the commands **set** and **unset.** A switch option is either on or off (set or unset). String options are strings of characters 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 looks 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 following sections describe the options available for the **mail** system.

askcc

The **askcc** switch causes prompting for additional carbon copy recipients when you finish composing a message. Press RETURN to accept the current list. Pressing the INTERRUPT key prints

```
interrupt (continue)
```

to continue editing the message.

dot

The **dot** switch enables you to use a period (.) as an end-of-transmission character, as well as CONTROL-D. This option is available for those who are used to this convention when editing with **ed.**

metoo

When you use an alias to send a message to a group and the alias contains your name, you usually do not receive a copy of the message. Setting the **metoo** option causes the sender to receive a copy of the message.

nosave

By default, when you abort a message you are composing, it is saved in **dead.letter** in your home directory. Setting the **nosave** option prevents aborted messages from being saved.

autoprint

Setting the **autoprint** option causes the **delete** command to behave like **dp.** After deleting (or undeleting) a message, the next message in the list is automatically displayed.

chron

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

mchron

Setting **mchron** displays messages in chronological order but lists them in the opposite order, i.e., most recent first. This is useful if you keep a large number of messages in your mailbox and wish to list the headers of the messages most recently received but read them in chronological order.

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.

EDITOR

The **EDITOR** string contains the path name of the text editor to use in the **edit** command and **~e** escape. If not defined, then **ed** is used. For example

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

escapes to **ed.**

VISUAL

The **VISUAL** string contains the path name of the text editor used in the **visual** command and **~v** escape. By default **vi** is the editor used. For example

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

escapes to **vi.**

SHELL

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

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

escapes to the Bourne shell (sh).

escape

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

```
set escape = *
```

causes the asterisk to become the new compose escape character.

page

The **page** string causes messages to be displayed in pages of size *n* lines. For example, to set the page length to 10 lines, type

```
set page = 10
```

You are prompted with a question mark between pages. Pressing RETURN causes the next page of the current message to be printed. By default this paging feature is turned off.

record

The **record** string sets the path name of the file used to record all outgoing mail. If not defined, then outgoing mail is not copied and saved. For example

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

causes all outgoing mail to be automatically appended to the file **/usr/john/recordfile.**

toplines

The **toplines** string sets the number of lines of a message to be printed out with the **top** command. By default, this value is 5. For example

```
set toplines = 10
```

causes 10 lines of each message to be printed out when the **top** command is used.

ignore

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

alias

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**), any mail would be sent to John, Paul, George, and Ringo. 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 **.mailrc** file so that you don't have to redefine them each time you invoke **mail.**

Using Advanced Features

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

Command Line Options

One very useful command line option is the **-s** "subject" switch. 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:

-R	Makes the **mail** session "read-only", preventing alteration of the mail being read.
-b	Adds the **blind carbon copy** field to the message header.
-c	Adds the **carbon copy** field to the message header.
-r	Adds the **return-receipt to** field to the message header.
-u	Reads in *user's* mail.

The XENIX mail system also enables 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

```
mail -f
```

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

Using mail as a Reminder Service

Besides sending and receiving mail, you can use **mail** as a reminder service. Several XENIX commands have this idea built in to them. For example, the XENIX **lpr** command's **-m** switch causes mail to be sent to the user after files have been printed on the line printer. XENIX automatically examines the file named "calendar" in each user's home directory and looks for lines containing either today or tomorrow's date. These lines are sent by **mail** as a reminder of important events.

When programming in the shell command language, you can use **mail** to signal job completion. For example, you might place the following line in a shell procedure:

```
biglongjob echo "biglongjob done" | mail self
```

You can also create a logfile that you want to mail to yourself. For example, you might have a shell procedure that looks like this:

```
dosomething >logfile mail self <logfile
```

Handling Large Amounts of mail

Eventually, you will face the problem of dealing with an accumulation of messages in your user mailbox. You can employ a number of strategies to handle this. If your mailbox file becomes large, periodically examine its contents to decide whether messages are still relevant. For long messages, consider using summaries.

Even the above measures are usually not enough to organize the many messages you are likely to receive. One effective approach is to save **mail** in files organized by sender, by topic, or by a combination of the two. Create these files in a separate **mail** directory; you can access these mailbox files with the **mail -f** *filename* switch.

Quick Reference

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

mail Files and Programs

The following is a list of the programs and files that make up the XENIX **mail** system:

/usr/bin/mail	Mail program
/usr/lib/mail/mailrc	Mail system initialization file
/usr/spool/mail/*	System mailbox files
/usr/*name*/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	Systemwide aliases
/usr/lib/mail/aliashash	Program to produce the **/usr/lib/main/alias.hash** file from the **/usr/lib/mail/aliases** file.

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

Command Summary

Given below are the name and syntax for each command, its abbreviated form (in brackets), and a short description. Many commands have optional arguments; most can be executed without any arguments at all. In particular, commands that take a message-list argument will default to the current message if no message-list is given. In the following descriptions, italics indicate arguments to commands or compose escapes. The vertical bar indicates selection and is used to separate the arguments from which you may select. All other text should be read literally.

RETURN	Prints the next message.
+*n*	[+] With no *n* argument, goes to the next message and prints it. If given a numeric argument *n*, goes to the *n*th 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 *n*th 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	Prints systemwide aliases for users.
alias *name users*	[a] Uses *name* as an alias for the group of *users*. With no *name* arguments, prints all currently defined aliases. With one argument, prints the *users* aliased by the given *name* argument.
cd *directoryname*	[c] Changes the user's working directory to the specified directory. If no directory is given, then changes to the user's home directory.
delete *mesg-list*	[d] Deletes each message in the given message-list.
dp *mesg-list*	Deletes the current message and prints the next message.
echo	Expands shell metacharacters.
edit *mesg-list*	[e] Takes the given message-list and points the text editor at each message in turn. On return to command mode, the edited message is read back in. See also the **visual** command.
exit[!]	[x] Immediately returns to the shell without modifying the system mailbox, the user mailbox, or a file specified with the **-f** switch.
file	[fi] Prints the name of the mailbox file.
forward *mesg-num user-list*	[f] Takes a *user-list* argument and forwards the current message to each name. The message sent to each is indented and shows that the sender has passed it on. The *mesg-num* argument is optional and is used to forward the numbered message instead of the default message.
Forward *mesg-num user-list*	[F] Same as **forward** except that the message is not indented.

headers *+n* | *-n* | *mesg-list*
: [h] With no argument, lists the current range of headers, which is an 18-message group. If a plus (+) argument is given, then the next 18-message group is 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.

hold *mesg-list*
: [ho] Takes a message-list and marks each message to be saved in the user's system mailbox instead of in **mbox.**

list
: Prints list of **mail** commands.

lpr *mesg-list*
: [l] Prints each of the messages in the required message-list on the line printer. Messages are piped through **pr** before being printed.

mail *user-list*
: [m] Takes an optional user-list argument and sends mail to each name after entering compose mode.

mbox *mesg-list*
: [mb] Marks messages given in the message-list argument to be saved in the user mailbox when a **quit** is executed. Message headers contain an initial letter "M" to show that they are to be saved.

move *mesg-list mesg-num*
: Places the messages specified in *mesg-list* after the message specified in *mesg-num*. If *mesg-num* is 0, *mesg-list* moves to the top of the mailbox.

print *mesg-list*
: [p] Takes a message-list and prints each message on the user's terminal.

quit
: [q] Terminates the **mail** session, retaining all nondeleted, unsaved messages in the system mailbox. 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 executing a **quit** while editing a mailbox file with the **-f** flag, the mailbox file is rewritten and the user returns to the shell.

reply *mesg-list*
: [r] Takes a message-list and sends mail to each message author just like the **mail** command.

Reply *mesg-list*
: [R] Identical to the **reply** command except that replies are also sent to other users in the **To:** and those named in the **Cc:** field.

save *mesg-list filename*
: [s] Takes an optional message-list and a file name 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 *mesg-list*	[si] Takes a message-list and prints the size in characters of each message.
source *file*	[so] Reads and executes **mail** commands from the given file.
top	[t] Takes a message-list and prints the top five lines. The number of lines printed is set by the variable **toplines.**
string *string mesg-list*	Searches for *string* in *mesg-list*. Ignores case in search.
undelete *mesg-list*	[u] Takes a message-list and marks each one as *not* being deleted. Each message in the list must have been previously deleted.
unset *options*	[uns] Takes a list of option names and discards their remembered values; this is the opposite of **set.**
visual *mesg-list*	[v] Takes a message-list and invokes the **vi** editor on each one.
write *mesg-list filename*	[w] Writes the message bodies of messages given by the message-list to the file given by file name.

Compose Escape Summary

The compose escapes listed below 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.

~~*string*	Inserts *string* in a message, prefaced by a single tilde (~).
~?	Displays **help** screen for compose escapes.
~.	Terminates text entry and mails message.
~!*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] Prints list of systemwide aliases.
~Alias *users*	[~A] Prints systemwide aliases for *users*.
~bcc *name ...*	[~b] Adds the given name(s) to the **Bcc:** field.
~cc *name ...*	[~c] Adds the given name(s) to the **Cc:** field.
~dead	[~d] Reads the file **dead.letter** from your home directory into the message.
~editor	[~e] Invokes the line editor on the message being sent. Exiting the editor returns the user to compose mode.
~headers	[~h] Edits the message heading fields by printing each one in turn and allowing the user to modify each field.
~message *mesg-list*	[~m] Reads the named messages into the message being sent, shifted right one tab. If no messages are specified, reads the current message.
~Message *mesg-list*	[~M] Same as **~message** except with no right shift.
~print	[~p] Prints the message buffer prefaced by the message heading.
~quit	[~q] Aborts the message being sent and copies 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 name(s) to the **Return-receipt-to:** field.
~shell	[~sh] Invokes the Bourne shell.
~subject *string*	[~s] Causes the named string to become the current subject field.
~to *name ...*	[~t] Adds the given name(s) 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 path name, a number, or a single character. Options are summarized below.

askcc	Causes prompting for additional carbon copy recipients at the end of each message. Pressing RETURN retains the current list.
autoprint	Causes the **delete** command to behave like **dp.** 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 blank line to act as the end-of-transmission character; CONTROL-D also works.
EDITOR=	Path name of the text editor to use in the **edit** command and **~e** escape. If not defined, then **ed** is used.
escape *char*	Sets *char* as the character to use as compose escape character.
ignore	Causes interrupt signals from the 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	Causes senders name to be included in alias expansions.
nosave	Prevents saving of the message buffer in the file **dead.letter** in the home directory, after two interrupts or a **~q** escape.
page=*n*	Specifies number of lines per page (*n*) when displaying messages.
quiet	Suppresses the printing of the version when **mail** is first invoked.
record=	Sets the path name of the file used to record all outgoing mail. If not defined, then outgoing mail is *not* copied.
SHELL=	Path name of the shell to use in the **!** command and the ~! escape. The Bourne 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.
VISUAL=	Path name of the text editor to use in the **visual** command and **~v** escape. The default is for the **vi** editor.

intel®

CHAPTER 7
bc: A CALCULATOR

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 that can input, output, and perform calculations 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 be compatible with the C language. A small collection of library functions is also available, including sine, cosine, arctangent, 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 enables use of decimal point notation. Provision is also 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 handled depends on the amount of storage available on the machine, so manipulation of numbers with many hundreds of digits is possible.

Invoking bc and Exiting

To invoke the **bc** calculator, type

```
bc
```

and press RETURN. You can perform calculations as described in this chapter.

When you are ready to exit from **bc**, press CONTROL-D until the command prompt appears ($ is the default for the standard Bourne shell).

Scaling Quantities

Before you begin to do calculations, you need to understand how **bc** deals with the number of decimal places. Numbers can have up to 99 decimal digits after the decimal point and the fractional part is retained in further computations. The number of digits after the decimal point is referred to as its scale.

A special internal quantity called **scale** is used to determine the scale of calculated quantities (as opposed to input quantities). The contents of **scale** can be no greater than 99 and no less than 0. To check **scale,** type

```
scale
```

and press RETURN. To see how **scale** affects output, consider this example:

```
scale=0
64/8
```

In this example, **scale** is set to 0, then 64 is divided by 8. The output has no decimal places and is displayed as

```
8
```

For comparison, consider this example:

```
scale=5
64/8
```

In this example, **scale** is set to 5, then 64 is divided by 8. The output has five decimal places and is displayed as

```
8.00000
```

When two 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; the result is never truncated.
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. Subject to these two restrictions, the scale of the result is set as close to the contents of **scale** as possible.
Division	The scale of a quotient is the contents of the internal quantity, **scale.**
Modulo	The scale of a remainder is the scale of the result of `a-a/b*b`

Exponentiation	The result of an exponentiation is scaled as if the implied multiplications were performed. An exponent must be an integer or you get a warning message and the exponent is treated as 1.
Square Root	The scale of a square root is set to the maximum of the scale of the argument and the contents of **scale.**

Basic Arithmetic Operations

This section discusses basic arithmetic operations.

Operators

The following operators are used in making calculations:

Operator	Meaning	Example (with **scale=0**)	Result
+	addition	142857 + 285714	428571
-	subtraction	4765 - 325	4440
*	multiplication	125 * 2	250
/	division	5000/5	1000
%	modulo [a%b is a-a/b*b]	15%7	1
^	exponentiation	2^4	16

Division produces a result truncated toward zero in the specified scale. Division by zero produces a warning message and a result of 0.

There is also a built-in square root function whose result is truncated to match **scale.** In this example, **scale** is 0, and the lines

```
scale=0
sqrt(191)
```

produce the output

```
13
```

In this example, **scale** is 4, and the lines

```
scale=4
sqrt(191)
```

produce the output

```
13.8202
```

Expressions

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

142857 + 285714

and press RETURN, **bc** responds immediately with the line

```
428571
```

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 (^) being 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 above operations are performed from left to right, except exponentiation, which is performed from right to left. **bc** shares with FORTRAN and C the convention that a/b*c is equivalent to (a/b)*c. Thus the two expressions

a^b^c

a^(b^c)

are equivalent, as are the two expressions

a*b*c

(a*b)*c

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 internal quantity **scale** can be used in expressions just like other variables. The line

scale = scale + 1

increases the value of **scale** by one.

Registers

You may use 26 internal storage registers to hold numbers. The name of a register must be a single lowercase letter, a-z.

This statement gives a register named "x" a value of 10:

```
x = 10
```

The result is not displayed on the screen, but you can see it by giving the register's name; in this case

```
x
```

causes this to appear:

```
10
```

You can also assign the value of an expression to a register. In this example, a register named "k" is given a value of 5, then 10 is added to the register:

```
k = 5
k = k + 5
```

In these lines, the value of the expression on the right of the equals sign is computed and stored in the register named on the left of the equals sign. Again, the result is not displayed, but you can see the register's contents by giving the register's name.

Registers may also be referred to as simple variables.

Advanced Features of bc

Specifying Input and Output Bases

bc has special internal quantities called **ibase** (input base) and **obase** (output base). **ibase** (**base** can be used in its place and means the same thing) is initially set to 10 and determines the base used for interpreting numbers read by **bc.** For example,

```
ibase = 8
11
```

produces

```
9
```

bc will now do octal-to-decimal conversions. However, recognize that you can't change the input base back to decimal by typing

```
ibase = 10
```

because the number 10 is interpreted as octal. 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. Thus, the statement

```
ibase = A
```

changes the base back to decimal, regardless of the current input base. Negative and large positive input bases can be used; 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 decimal 10. The lines

```
obase = 16
1000
```

produce

```
3E8
```

This is interpreted as a three-digit hexadecimal number.

Very large output bases may be used. In this case, each digit is expressed as a decimal number and the decimal numbers are separated by a space. 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, 1, and 0, are handled correctly.

Very large numbers are automatically split across lines with 70 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.

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

Any fractional part of **ibase** or **obase** is truncated. For example, if you specify **ibase=3.5**, then ask for **ibase**, the response is 3.

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

Using Functions

You can define 26 functions, named a-z, and you can use the same name for a function and a register. For example, you can have a function named "m" and a register named "m".

The line

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

begins the definition of a function named "a" 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 (}). A function returns control 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(*expression*)

In the first form, the returned value of the function is 0; in the second, it is the value of the expression in parentheses. For example, this statement returns the value of "r+4":

```
return(r+4)
```

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

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

Functions may contain only one **auto** statement, 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, except that they are given a value on entry to the function.

This example defines a function named "a":

```
define a(x,y){
        auto z
        z = x*y
        return(z)
}
```

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

To call a function, give 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 is used. For example, if function "a" is defined as shown above, then the line

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

would produce

```
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, if you have defined a function named "b" with no arguments, you would call it with

```
b ( )
```

Using Subscripted Variables

A single lowercase 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 can be used in **bc.** The names of arrays can be the same as 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 b[ ]

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

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 (a collection of statements enclosed in braces). They are written as follows:

if (*relation*) *statement*
while (*relation*) *statement*
for (*expression1*; *relation*; *expression2*) *statement*

if (*relation*) {*statements*}
while (*relation*) {*statements*}
for (*expression1*; *relation*; *expression2*) {*statements*}

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:

```
< > <= >= == !=
```

Note that a double equal sign (==) stands for "equal to" and an exclamation-equal sign (!=) stands for "not equal to". The meaning of the remaining relational operators is their normal arithmetic and logical meaning.

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

The **while** statement causes execution of its range repeatedly 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** 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 examples illustrate 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(5)** produces 120, which is the factorial of 5.

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
      }
}
```

With function "e" defined as shown, the line **e(0)** produces

```
1.00000000000000000000
```

and the line **e(1)** produces

```
2.71828182845904523526
```

Using Other Language Features

Some important language features are listed below.

- Statements are usually typed one to a line, but you can type several statements on a line if they are separated by semicolons.

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

  (x=y+17)

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

  The following uses the value of an assignment statement even when it is not 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. (Note: nothing is displayed if you just type this line by itself, since it is a partial example.)

- The following constructions can be used in **bc**:

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

  Even if you don't intend to use these constructions, if you type one inadvertently, something legal but unexpected may happen. Be aware that in some of these constructions spaces are significant. For example, in the constructions "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 "*/". They may span lines.

- When you invoke **bc**, you can load a library of math functions by using the **-l** option as follows:

  bc -l

  This library sets **scale** to 20 by default. The library functions are

s(x)	Compute the sine of x (x is input in radians).
c(x)	Compute the cosine of x (x is input in radians).
a(x)	Compute the arctangent of x (x is input in radians).
l(x)	Compute the natural logarithm of x.
e(x)	Compute the exponent of x (e^x).
j(n,x)	Compute the Bessel function of n and x.

- If you type

  bc *filename*

  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 blanks, tabs, or comments. Newline characters or semicolons separate statements.

Comments — Comments are introduced by /* and are terminated by */. They may span lines.

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

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

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 order in which they are evaluated is as follows:

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

There are several types of expressions:

named expressions	Named expressions are 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.
identifiers	Simple identifiers are named expressions. They have an initial value of zero.
array-name [expression]	Array elements are named expressions. They have an initial value of zero.
scale, ibase, and obase	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. **ibase** and **obase** are the input and output number radixes respectively. Both **ibase** and **obase** have initial values of 10.

constants	Constants are primitive expressions that evaluate to themselves.
parenthetic expressions	An expression surrounded by parentheses is a parenthetic expression. The parentheses may be used to alter normal operator precedence.
function calls	Function calls are expressions that return values.

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. Three built-in functions are listed below:

sqrt(*expr*)	The result is the square root of the expression and 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.

Exponentiation Operators

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))
```

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.**

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 assignment operators listed below assign values to the named expression on the left side.

named_expr=expr	This expression results in assigning the value of the expression on the right to the named expression on the left.
named_expr=+expr	The result of this expression is equivalent to *named_expr=named_expr+expr*.
named_expr=-expr	The result of this expression is equivalent to *named_expr=named_expr-expr*.
*named_expr=*expr*	The result of this expression is equivalent to *named_expr=named_expr*expr*.
named_expr=/expr	The result of this expression is equivalent to *named_expr=named_expr/expr*.
named_expr=%expr	The result of this expression is equivalent to *named_expr=named_expr%expr*.
named_expr=^expr	The result of this expression is equivalent to *named_expr=named_expr^expr*.

Relational Operators

Unlike all other operators, the relational operators are only valid as the object of an **if** or **while** statement, or inside a **for** statement. These operators are listed below:

expr<expr	The first expression is less than the second.
expr>expr	The first expression is greater than the second.
expr<=expr	The first expression is less than or equal to the second.
expr>=expr	The first expression is greater than or equal to the second.
expr==expr	The first expression is equal to the second.
expr!=expr	The first expression is not equal to the second.

Storage Classes

bc has only two storage classes: global and automatic (local). Only identifiers local to a function need to 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. Therefore, they do not retain values between function calls. Note that **auto** arrays are specified by the array name, followed by empty square brackets.

Automatic variables in **bc** do not work the same way as 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. Built-in statements include **auto, break, define, for, if, quit, return,** and **while.** Each kind of statement is discussed below:

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 can be grouped together and used when one statement is expected by surrounding them with curly braces ({ }).

Quoted string statements

For example

"*string*"

prints the string inside the quotation marks. A carriage return is allowed within the quotation marks.

Built-in statements

The syntax for each built-in statement is given below:

auto

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*]

break

The **break** statement causes termination of a **for** or **while** statement. Syntax for the **break** statement is

break

define

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*}

for

The **for** statement is the same as

```
first-expression
while (relation) {
    statement
    last-expression
}
```

All three expressions must be present. Syntax of the **for** statement is

for (*expression*; *relation*; *expression*) *statement*

if

The **if** statement is executed if the relation is true. The syntax is

if (*relation*) *statement*

quit

The **quit** statement stops execution of a **bc** program and returns control to XENIX when 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 CONTROL-D is the same as typing **quit.** The syntax of the **quit** statement is

quit

return

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*)

while

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*

intel®

APPENDIX A RELATED PUBLICATIONS

Intel Publications

Copies of the following publications can be ordered from

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

XENIX R. 3.4 Reference Library: Basic System

Overview of the XENIX 286 Operating System, Order Number 174385 -- XENIX history, XENIX uses, basic XENIX concepts, and an overview of other XENIX manuals.

XENIX 286 User's Guide, Order Number 174387 -- a brief survey of common commands plus full chapters on the **ed** text editor, the **vi** text editor, electronic **mail**, the Bourne shell (**sh**), and the **bc** calculator.

XENIX 286 Visual Shell User's Guide, Order Number 174388 -- a XENIX command interface ("shell") that replaces the standard command syntax with a menu-driven command interpreter.

XENIX 286 Installation and Configuration Guide, Order Number 174386 -- how to install XENIX on your hardware and tailor the XENIX configuration to your needs.

XENIX 286 System Administrator's Guide, Order Number 174389 -- how to perform system administrator chores such as adding and removing users, backing up file systems, and troubleshooting system problems.

XENIX 286 Communications Guide, Order Number 174461 -- installing, using, and administering XENIX networking software.

XENIX 286 Reference Manual, Order Number 174390 -- all commands in the XENIX 286 Basic System.

XENIX R. 3.4 Reference Library: Extended System

XENIX 286 Programmer's Guide, Order Number 174391 -- XENIX 286 Extended System commands used for developing and maintaining programs.

XENIX 286 C Library Guide, Order Number 174542 -- standard subroutines used in programming with XENIX 286, including all system calls.

XENIX 286 Device Driver Guide, Order Number 174393 -- how to write device drivers for XENIX 286 and add them to your system.

XENIX 286 Text Formatting Guide, Order Number 174541 -- XENIX 286 Extended System commands used for text formatting.

Other XENIX Publications

XENIX Networking Software Installation and Configuration Guide, Order Number 135146 -- installing, configuring, and administering the XENIX OpenNET™ network.

XENIX Networking Software User's Guide, Order Number 135147 -- user's and programmer's reference to the XENIX OpenNET™ network.

intel®

INDEX

intel®

XENIX
User's Gu
174387-

REQUEST FOR READER'S COMMENTS

Intel's Technical Publications Departments attempt to provide publications that meet the needs of
Intel product users. This form lets you participate directly in the publication process. Your commer
will help us correct and improve our publications. Please take a few minutes to respond.

Please restrict your comments to the usability, accuracy, organization, and completeness of tl
publication. If you have any comments on the product that this publication describes, please conta
your Intel representative. If you wish to order publications, contact the Literature Department.

1. Please describe any errors you found in this publication (include page number).

____

____

____

____

2. Does this publication cover the information you expected or required? Please make suggestio
for improvement.

____

____

____

____

3. Is this the right type of publication for your needs? Is it at the right level? What other types
publications are needed?

____

____

____

____

4. Did you have any difficulty understanding descriptions or wording? Where?

____

____

____

5. Please rate this publication on a scale of 1 to 5 (5 being the best rating). __________

NAME ______________________________ DATE __________

TITLE ______________________________

COMPANY NAME/DEPARTMENT ______________________________

ADDRESS ______________________________

CITY ______________ STATE (COUNTRY) ______________ ZIP CODE __________

Please check here if you require a written reply. ☐

'E'D LIKE YOUR COMMENTS . . .

ıis document is one of a series describing Intel products. Your comments on the back of this form will ·lp us produce better manuals. Each reply will be carefully reviewed by the responsible person. All ımments and suggestions become the property of Intel Corporation.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 79 BEAVERTON, OR

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
OMS Technical Publications, MS: HF2-52
5200 N.E. Elam Young Parkway
Hillsboro, Oregon 97124-9987

intel

INTEL CORPORATION
3065 Bowers Avenue
Santa Clara, California 95051
(408) 987-8080

JAPAN
Intel Japan K.K.
5-6 Tokodai Toyosato-machi
Tsukuba-gun, Ibaraki-ken 300-26
Japan

FRANCE
Intel
5 Place de la Balance
Silic 223
94528 Rungis Cedex
France

UNITED KINGDOM
Intel
Piper's Way
Swindon
Wiltshire, England SN3 1RJ

WEST GERMANY
Intel
Seidstrasse 27
D-8000 Munchen 2
West Germany

SOFTWARE
Printed in U.S.A.
0920/10K/0186/PTW/AD