XENIX® System V

Operating System

User's Guide

This document was typeset with an IMAGEN® 8/300 Laser Printer.

SCO Document Number: XG-4-1-87-4.0

Acknowledgments

This manual builds on the writing of many others. In many cases, the content here is identical, in whole or in part, to papers and manuals written at Bell Laboratories. In particular, Chapters 2 is adapted from papers written by Brian Kernighan. Chapter 3 is adapted from a document written by Kurt Shoens at the University of California at Berkeley. Chapter 4 is adapted from documents written by G. A. Snyder, J. R. Mashey, and S. R. Bourne. Chapter 6 is from a paper written by Robert Morris and Lorinda Cherry. The work of those mentioned above, and countless others, is gratefully acknowledged.

Contents

A ed

Chapter 1

Introduction

1.1 Overview

This guide introduces several basic XENIX facilities, including **mail**, text editors, and powerful operating environments called "shells."

1.2 About This Guide

This guide is organized as follows:

Chapter 1, "Introduction," gives an introduction and overview of the XENIX system. It also gives a list of conventions used throughout this guide.

Chapter 2, "vi," explains how to use the screen editor, **vi**(C).

Chapter 3, "mail," describes the XENIX **mail** (C) facility and explains how to send and receive **mail.**

Chapter 4, "The Shell," describes use of the shell, (sh (C)), command interpreter and how to write procedures that can be executed by **sh.**

Chapter 5, "bc: A Calculator," explains how to use **bc**(C) a sophisticated calculator program.

Chapter 6, "Building a Communications System," explains how to set up a system to permit communication between XENIX and/or UNIX systems using dial-up communication lines.

Chapter 7, "The C-Shell," describes how to use **csh**(C). It covers the syntax and function of C-shell, (**csh** (C)), commands and features, and how to create shell procedures.

Chapter 8, "Using The Visual Shell," describes the use and behavior of the Visual Shell, (**vsh** (C)), which is a menu-driven XENIX shell. This chapter assumes the reader is familiar with some general XENIX concepts, but vsh can be used by first-time users.

Appendix A "ed" explains how to use the editor, **ed**(C).

1.3 Where To Find More Information

This guide does not attempt to give information about installing, managing, and maintaining the system, nor does it discuss document preparation, software development, or many of the specialized utilities available in other XENIX system products.

You can find more information on these subjects in the guides found in the following binders:

"Run Time Environment"

The XENIX Installation Guide describes how to install and set up the XENIX system on your computer.

Introduction to XENIX introduces the XENIX system by presenting key concepts in a tutorial format.

The XENIX Operations Guide explains how to manage and maintain the system.

Hardware Dependent Reference serves as a comprehensive command reference, for Hardware Dependent (HW) commands.

"User's Reference"

The XENIX User's Reference serves as a comprehensive, hardware independent, Operating System, command reference. A concise but complete description of each command is available here. It includes manual pages for Commands(C), Miscellaneous(M), and File Formats(F).

"Programmer's Reference"

The XENIX Programmer's Reference serves as a comprehensive Development System command reference. It includes the manual page reference sections for Programming Commands(CP), System Calls(S), and DOS Routines(DOS). This guide is part of the optional XENIX Development System.

"Programmer's Guide I"

The XENIX Programmer's Guide discusses how to use the programming tools available in the XENIX programming environment. This guide is part of the optional XENIX Development System.

"*C Language Reference*" describes the various elements of the C programming language. It is intended as a reference for programmers already familiar with C or another language. This guide is part of the optional XENIX Development System.

"Programmer's Guide II"

C User's Guide discusses writing C language programs that interface to the XENIX operating system. It provides reference to system calls, subroutines, and file formats. This guide is part of the optional XENIX Development System.

C Library Guide provides information about the standard include files, tells how to build user interfaces for C programs, provides a full description of error messages, and provides information on cross development including a list of library routines common to both XENIX and DOS. This guide is part of the optional XENIX Development System.

Macro Assembler (MASM) User's Guide explains how to create and debug assembly language programs using the Macro Assembler, MASM, (**masm** (CP)). This guide is part of the optional XENIX Development System.

Macro Assembler (MASM) Reference Manual describes the usage and input syntax of the Macro Assembler, MASM, (**masm** (CP)). This guide is part of the optional XENIX Development System.

"Text Processing Guide"

The XENIX Text Processing Guide explains how to use the text processing and text formatting tools and includes the manual pages for Text Commands(CT). It is a part of the optional XENIX Text Processing System.

1.4 Notational Conventions

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

Initial Capitals	Initial Capitals indicate the name of a command or mode. When a command is introduced it is followed by the keystroke that invokes it, (i.e. the Insert (i) command).
boldface	Boldface indicates a command, option, flag, or program name to be entered as shown. Keystrokes are boldfaced when they indicate a command to enter as shown, (i.e. enter the **i** command and press **RETURN**).
	Boldface indicates the name of a library routine.

(To find more information on a given library routine consult the "Alphabetized List" in your XENIX *Reference Manual* for the manual page that describes it.)

italics — Italics indicate a filename. This pertains to library include filenames (i.e. *stdio.h*), as well as, other filenames (i.e. */etc/ttys*).

Italics indicate a placeholder for a command argument. When entering a command, a placeholder must be replaced with an appropriate filename, number, or option.

Italics indicate a specific identifier, supplied for variables and functions, when mentioned in text.

Italics indicate a reference to part of an example.

Italics indicate emphasized words or phrases in text.

[ ] — Brackets indicate that the enclosed item is optional. If you do not use the optional item, the program selects a default action to carry out.

Brackets indicate the position of the cursor in text examples.

... — Ellipses indicate that you can repeat the preceding item any number of times.

Vertical ellipses indicate that a portion of a program example is omitted.

" " — Quotation marks indicate the first use of a technical term.

Quotation marks indicate a reference to a word rather than a command.

Chapter 2

vi: A Text Editor

2.1 Introduction

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

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

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

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

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

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

2.2 Demonstration

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

Before you start the demonstration, make sure that your terminal has been properly set up. See section 2.5.1, "Setting the Terminal Type", for more information about setting up your terminal for use with **vi**.

2.2.1 Entering the Editor

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

```
vi temp
```

Your screen will look like this:

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

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

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

> The line containing the cursor is called the *current line.*

The lines containing tildes are not part of the file: they indicate lines on the screen only, not real lines in the file.

2.2.2 Inserting Text

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

```
i
```

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

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

Press the **ESCAPE** key (abbreviated **ESC**) when you are finished.

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

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

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

2.2.3 Repeating a Command

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

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

2.2.4 Undoing a Command

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

u

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

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

Now enter:

u

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

2.2.5 Moving the Cursor

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

h Left

l Right

k Up

j Down

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

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

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

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

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

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

2.2.6 Deleting

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

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

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

1G

At first, your file should look like this:

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

To delete the first line, enter:

dd

Your file should now look like this:

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

Delete the word the cursor is sitting on by entering:

dw

After deleting, your file should look like this:

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

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

x

This leaves:

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

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

d$

Your file looks like this:

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

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

d0

This leaves a single space on the line:

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

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

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

```
Files contain text.
Text contains lines.
```

Press **ESC** to go back to Command mode.

2.2.7 Searching for a Pattern

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

```
H
```

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

```
/char
```

Do not press RETURN yet. Your screen should look like this:

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

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

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

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

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

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

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

2.2.8 Searching and Replacing

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

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

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

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

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

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

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

Press **RETURN**. Your screen should look like this:

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

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

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

u

Your screen now looks like this:

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

2.2.9 Leaving vi

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

    :x

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

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

Then the XENIX prompt appears.

2.2.10 Adding Text From Another File

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

    vi practice

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

    :r temp

Your file should look like this:

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

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

2.2.11 Leaving vi Temporarily

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

```
:!date
```

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

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

:!date
Mon Jan 9 16:33:37 PST 1985
[Hit return to continue]_
```

2.2.12 Changing Your Display

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

To turn on automatic line numbering, enter:

```
:set number
```

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

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

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

```
:set all
```

and pressing **RETURN**. Setting these options is described in section 2.5 "Setting Up Your Environment", but it is important that you be aware of their existence. Depending on what you are working on, and your own preferences, you will want to alter the default settings for many of these options.

2.2.13 Canceling an Editing Session

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

```
:q!
```

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

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

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

2.3 Editing Tasks

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

2.3.1 How to Enter the Editor

There are several ways to begin editing, depending on what you are planning to do. This section describes how to start, or "invoke" the editor with one filename. To invoke **vi** on a series of files, see section 2.3.24, "Editing a Series of Files".

With a Filename

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

vi *filename*

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

At a Particular Line

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

vi +100 *filename*

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

At a Particular Word

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

vi +/*word filename*

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

vi +/contain temp

2.3.2 Moving the Cursor

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

Moving the Cursor By Characters: h,j,k,l,SPACE,BKSP

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

4h

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

Fp

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

fp

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

Tp

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

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

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

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

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

> No, I didn’t know he had returned.

If and you press:

> 6w

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

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

> 6W

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

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

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

> 4b

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

Enter:

4B

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

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

Moving the Cursor by Lines

Forward: j, Ctrl-n, +, RETURN, LINEFEED, $

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

6+

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

Lines contain characters.
Text contains lines.

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

4$

Backward: k, Ctrl-p

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

4k

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

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

2.3.3 Moving Around in a File: Scrolling

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

Scrolling Up Part of the Screen: Ctrl- u

Ctrl- u scrolls up one-half screen.

Scrolling Up the Full Screen: Ctrl- b

Ctrl- b scrolls up a full screen.

Scrolling Down Part of the Screen: Ctrl- d

Ctrl- d scrolls down one-half screen.

Scrolling Down a Full Screen: Ctrl- f

Ctrl- f scrolls down a full screen.

Placing a Line at the Top of the Screen: z

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

```
z
```

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

```
33z
```

Press **RETURN**, and line 33 scrolls to the top of the screen. For information on how to display line numbers, see section 2.5.5, "Displaying Line Numbers: number".

2.3.4 Inserting Text Before the Cursor: i and I

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

```
i
```

The "i" does not appear on the screen. Any text typed after the "i" becomes part of the file you are editing. To leave Insert mode and reenter Command mode, press **ESC**. For more explanation of modes in **vi**, see section 2.2.2, "Inserting Text".

Anywhere on a Line: i

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

At the Beginning of the Line: I

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

2.3.5 Appending After the Cursor: a and A

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

Anywhere on a Line: a

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

At the end of a Line: A

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

2.3.6 Correcting Typing Mistakes

If you make a mistake while you are typing, the simplest way to correct it is with the **BKSP** key. Backspace across the line until you have backspaced over the mistake, then retype the line. You can only do this, however, if the cursor is on the same line as the error. See sections 2.3.12 through 2.3.15 for other ways to correct typing mistakes.

2.3.7 Opening a New Line

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

You may also use the **RETURN** key to open new lines above and below the cursor. To open a line above the cursor, move the cursor to the beginning of the line, press **i** to enter Insert mode, then press **RETURN**. (For information on how to move the cursor, see section 2.3.2, "Moving the Cursor".) To open a line below the cursor, move the cursor to the end of the current line, press **i** to enter Insert mode, then press **RETURN**.

2.3.8 Repeating the Last Insertion

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

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

For other methods of repeating an insertion, see section 2.3.8, "Repeating the Last Insertion", section 2.3.9, "Inserting Text From Other Files", and section 2.3.22, "Repeating a Command".

2.3.9 Inserting Text From Other Files

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

:r *filename*

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

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

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

2. Enter:

   :e *filename*

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

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

4. Enter:

   mk

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

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

   "ay'k

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

6. Enter:

   :e#

   to return to your previous file. (For more information about this command, see section 2.3.25, "Editing a New File Without Leaving the Editor".) Move the cursor to the line above the place you want the new text to appear, then enter:

   "ap

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

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

You may also delete text into a buffer, then insert it in another place. For information on this type of deletion and insertion, see section 2.3.17, "Moving Text".

Copying Lines From Elsewhere in the File

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

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

```
:nu
```

and press **RETURN**. The line number and the text of that line are displayed on the status line. To find out the destination line number, move the cursor to the line above where you want the copied text to appear and repeat the **:nu** command. You can also make line numbers appear throughout the file with the **linenumber** option. For information on how to set this option, see section 2.5.5, "Displaying Line Numbers: number". The following example uses the **number** option to display line numbers in a file.

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

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

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

The result is:

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

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

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

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

   "ayy

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

2. Enter the following line:

   "ap

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

Your screen looks like this:

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

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

"a3yy

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

2yw

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

10yl

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

y$

2.3.10 Inserting Control Characters into Text

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

2.3.11 Joining and Breaking Lines

To join two lines press:

    J

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

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

    r

then press **RETURN**.

2.3.12 Deleting a Character: x and X

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

    3x

To delete three characters preceding the cursor, enter:

    3X

2.3.13 Deleting a Word: dw

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

```
3dw
```

2.3.14 Deleting a Line: D and dd

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

```
3dd
```

Another way to delete several lines is to use a line-oriented command. To use this command it helps to know the line numbers of the text you wish to delete. For information on how to display line numbers, see section 2.5.5, "Displaying Line Numbers: number".

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

```
:200,250d
```

Press **RETURN**.

When the command finishes, the message:

```
50 lines
```

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

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

```
:.,$d
```

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

:.,+3d

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

:.,-3d

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

2.3.15 Deleting an Entire Insertion

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

2.3.16 Deleting and Replacing Text

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

Overstriking: r and R

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

rb

If a number is given before r, that number of characters is replaced with the next character entered. For example, to replace the character above the cursor, plus the next three characters, with the letter "b", enter:

4rb

Note that you now have four "b"s in a row.

The **R** command replaces as many characters as you enter. To end the replacement, press **ESC**. For example, to replace the second line in the following text with "Spelling is important.":

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

Move the cursor over the *T* in *Text*. Press R, then enter:

Spelling is important.

Press **ESC** to end the replacement. If you make a mistake, use the **BKSP** key to correct it. Your screen should now look like this:

```
Files contain text.
Spelling is important[.]
Lines contain characters.
Characters form words.
Words form text.
~
~
~
~
~
~
~
```

Substituting: s and S

The s command replaces a specified number of characters, beginning with the character under the cursor, with text you enter. For example, to substitute "xyz" for the cursor and two characters following it, enter:

3sxyz

The S command deletes a specified number of lines and replaces them with text you enter. You may enter as many new lines of text as you wish; S affects only how many lines are deleted. If no number is given, one line is deleted. For example, to delete four lines, including the current line, enter:

4S

This differs from the **R** command. The S command deletes the entire current line; the **R** command deletes text from the cursor onward.

Replacing a Word: cw

The **cw** command replaces a word with text you enter. For example, to replace the word "bear" with the word "fox", move the cursor over the "b" in "bear". Press:

cw

A dollar sign appears over the "r" in *bear*, marking the end of the text that is being replaced. Enter:

fox

and press **ESC**. The rest of "bear" disappears and only "fox" remains.

Replacing the Rest of a Line: C

The C command replaces text from the cursor to the end of the line. For example, to replace the text of the sentence:

Who's afraid of the big bad wolf?

from *big* to the end, move the cursor over the *b* in *big* and press:

C

A dollar sign ($) replaces the question mark (?) at the end of the line. Enter the following:

little lamb ?

Press **ESC**. The remaining text from the original sentence disappears.

Replacing a Whole Line: cc

The **cc** command deletes a specified number of lines, regardless of the location of the cursor, and replaces them with text you enter. If no number is given, the current line is deleted.

Replacing a Particular Word on a Line

If a word occurs several times on one line, it is often convenient to use a line-oriented command to replace it. For example, to replace the word *removing* with "deleting" in the following sentence:

In **vi**, removing a line is as easy as removing a letter.

Make sure the cursor is at the beginning of that line, and enter:

:s/removing/deleting/g

Press **RETURN**. This line-oriented command means "Substitute (s) for the word *removing* the word *deleting*, everywhere it occurs on the current line (g)". If you don't include a g at the end, only the first occurrence of *removing* is changed.

For more information on using line-oriented commands to replace text, see section 2.3.19, "Searching and Replacing."

2.3.17 Moving Text

To move a block of text from one place in a file to another, you can use the line-oriented **m** command. You must know the line numbers of your file to use this command. The **number** option displays line numbers. To set this option, press ESC to make sure you are in Command mode, then enter:

```
set number
```

Line numbers will appear to the left of your text. For more information on setting the **number** option, see section 2.5.5, "Displaying Line Numbers: number".

The following example uses the **number** option. For other ways to display line numbers, see section 2.3.29, "Finding Out What Line You're On".

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

To insert lines 2 and 3 between lines 4 and 5, enter:

```
:2,3m4
```

Your screen should look like this:

```
1 Files contain text.
2 Characters form words.
3 Text contains lines.
4 Lines contain characters.
5 [W]ords form text.
~
~
~
~
~
~
~
```

To place line 5 after line 2, enter:

:5m2

After moving, your screen should look like this:

```
1 Files contain text.
2 Characters form words.
3 [W]ords form text.
4 Text contains lines.
5 Lines contain characters.
~
~
~
~
~
~
~
```

To make line 4 the first line in the file, enter:

:4m0

Your screen should look like this:

```
1 [T]ext contains lines.
2 Files contain text.
3 Characters form words.
4 Words form text.
5 Lines contain characters.
~
~
~
~
~
~
~
```

You can also delete text into a temporary storage place, called a "buffer", and insert it wherever you wish. When text is deleted it is placed in a "delete buffer". There are nine "delete buffers".

The first buffer always contains the most recent deletion. In other words, the first deletion in a given editing session goes into buffer 1. The second deletion also goes into buffer 1, and pushes the contents of the old buffer 1 into buffer 2. The third deletion goes into buffer 1, pushing the contents of buffer 2 into buffer 3, and the contents of buffer 1 into buffer 2. When buffer 9 has been used, the next deletion pushes the current text of buffer 9 off the stack and it disappears.

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

Delete buffers are particularly useful when you wish to remove text, store it, and put it somewhere else. Using the following text as an example:

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

Delete the first line by entering:

dd

Delete the third line the same way. Now move the cursor to the last line in the example and press:

"1p

The line from the *second* deletion appears:

```
Text contains lines.
Characters form words.
Words form text.
[L]ines contain characters.
~
~
~
~
~
~
~
~
```

Now enter:

```
"2p
```

The line from the *first* deletion appears:

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

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

It is also possible to place text in named buffers. For information on how to create named buffers, see section 2.3.9, "Inserting Text From Other Files".

2.3.18 Searching: / and ?

You can search forward and backward for patterns in **vi**. To search forward, press the slash (/) key. The slash appears on the status line. Enter the characters you wish to search for. Press **RETURN**. If the specified pattern exists, the cursor will move to the first character of the pattern.

For example, to search forward in the file for the word "account", enter:

```
/account
```

Press **RETURN**. The cursor is placed on the first character of the pattern. To place the cursor at the beginning of the line above "account", for example, enter:

/account/-

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

/account/-2

To place the cursor two lines below "account", enter:

/account/+2

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

?account

To search for a pattern containing any of the special characters (. * \ [ ] ~ $ and ^), each special character must be preceded by a backslash. For example, to find the pattern "U.S.A.", enter:

/U\.S\.A\./

You can continue to search for a pattern by pressing:

n

after each search. The pattern is unaffected by intervening **vi** commands, and you can use **n** to search for the pattern until you enter a new pattern or quit the editor.

vi searches for exactly what you enter. If the pattern you are searching for contains an uppercase letter (for example, if it appears at the beginning of a sentence), **vi** ignores it. To disregard case in a search command, you can set the ignorecase option:

:set ignorecase

By default, searches "wrap around" the file. That is, if a search starts in the middle of a file, when vi reaches the end of the file it will "wrap around" to the beginning, and continue until it returns to where the search began. Searches will be completed faster if you specify forward or backward searches, depending on where you think the pattern is.

If you do not want searches to wrap around the file, you can change the "wrapscan" option setting. Enter:

:set nowrapscan

and press **RETURN** to prevent searches from wrapping. For more information about setting options, see section 2.5, "Setting Up Your Environment".

2.3.19 Searching and Replacing

The search and replace commands allow you to perform complex changes to a file in a single command. Learning how to use these commands is a must for the serious user of **vi**.

The syntax of a search and replace command is:

*g/pattern1/*s/[*pattern2*]/[*options*]

Brackets indicate optional parts of the command line. The **g** tells the computer to execute the replacement on every line in the file. Otherwise the replacement would occur only on the current line. The *options* are explained in the following sections.

To explain these commands we will use the example file from the demonstration run:

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

Replacing a Word

To replace the word "contain" with the word "are" throughout the file, enter the following command:

```
:g/contain /s//are /g
```

This command says "On each line of the file (g), find *contain* and substitute for that word (s//) the word *are*, everywhere it occurs on that line (the second g)". Note that a space is included in the search pattern for *contain*; without the space *contains* would also be replaced.

After the command executes your screen should look like this:

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

Printing all Replacements

To replace "contain" with "are" throughout the file, and print every line changed, use the **p** option:

```
:g/contain /s//are /gp
```

Press **RETURN**. After the command executes, each line in which "contain" was replaced by "are" is printed on the lower part of the screen. To remove these lines, redraw the screen by pressing **Ctrl-1**.

Choosing a Replacement

Sometimes you may not want to replace every instance of a given pattern. The **c** option displays every occurrence of *pattern* and waits for you to confirm that you want to make the substitution. If you press **y** the substitution takes place; if you press **RETURN** the next instance of *pattern* is displayed.

To run this command on the example file, enter:

```
:g/contain/s//are/gc
```

Press **RETURN**. The first instance of "contain" appears on the status line:

Files containtext.

Press **y**, then **RETURN**. The next occurrence of *contain* appears.

2.3.20 Pattern Matching

Search commmands often require, in addition to the characters you want to find, a context in which you want to find them. For example, you may want to locate every occurrence of a word at the beginning of a line. **vi** provides several special characters that specify particular contexts.

Matching the Beginning of a Line

When a caret(^) is placed at the beginning of a pattern, only patterns found at the beginning of a line are matched. For example, the following search pattern only finds "text" when it occurs as the first word on a line:

/^text/

To search for a caret that appears as text you must precede it with a backslash (\).

Matching the End of a Line

When a dollar sign ($) is placed at the end of a pattern, only patterns found at the end of a line are matched. For example, the following search pattern only finds "text" when it occurs as the last word on a line:

/text$/

To search for a dollar sign that appears as text you must precede it with a backslash (\).

Matching Any Single Character

When used in a search pattern, the period (.) matches any single character except the newline character. For example, to find all words that end with "ed", use the following pattern:

/.ed /

Note the space between the *d* and the backslash.

To search for a period in the text, you must precede it with a backslash (\).

Matching a Range of Characters

A set of characters enclosed in square brackets matches any single character in the range designated. For example, the search pattern:

/[a–z]/

finds any lowercase letter. The search pattern:

/[aA]pple/

finds all occurrences of "apple" and "Apple".

To search for a bracket that appears as text, you must precede it with a backslash (\).

Matching Exceptions

A caret (^) at the beginning of *string* matches every character *except* those specified in *string*. For example the search pattern:

[^a–z]

finds anything but a lowercase letter or a newline.

Matching the Special Characters

To place a caret, hyphen or square bracket in a search pattern, precede it with a backslash. To search for a caret, for example, enter:

/\^/

If you need to search for many patterns that contain special characters, you can reset the **magic** option. To do this, enter:

:nomagic

This removes the special meaning from the characters ., \, $, [and]. You can include them in search and replace commands without a preceding backslash. Note that the special meaning cannot be removed from the special characters star (*) and caret (^); these must always be preceded by a backslash in searches.

To restore *magic*, enter:

:set magic

For more information about setting options, see section 2.5, "Setting Up Your Environment".

2.3.21 Undoing a Command: u

Any editing command can be reversed with the Undo (**u**) command. The Undo command works on both screen-oriented and line-oriented commands. For example, if you have deleted a line and then decide you wish to keep it, press *u* and the line will reappear.

Use the following line as an example:

```
[T]ext contains lines.
~
~
~
~
~
~
~
~
~
~
~
```

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

```
Text [l]ines.
~
~
~
~
~
~
~
~
~
~
~
```

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

```
Text [c]ontains lines.
~
~
~
~
~
~
~
~
```

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

```
Text [l]ines.
~
~
~
~
~
~
~
~
~
~
```

It is important to remember that **u** only undoes the *last* command. For example, if you make a global search and replace, then delete a few characters with the **x** command, pressing **u** will undo the deletions but not the global search and replace.

2.3.22 Repeating a Command: .

Any screen-oriented **vi** command can be repeated with the Repeat (.) command. For example, if you have deleted two words by entering:

```
2dw
```

you may repeat this command as many times as you wish by pressing the period key (.). Cursor movement does not affect the Repeat command, so you may repeat a command as many times and in as many places in a file as you wish.

The Repeat command only repeats the last **vi** command. Careful planning can save time and effort. For example, if you want to replace a word that occurs several times in a file (and for some reason you do not wish to use a global command), use the **cw** command instead of deleting the word with the **dw** command, then inserting new text with the **i** command. By using the **cw** command you can repeat the replacement with the dot (.) command. If you delete the word, then insert new text, dot only repeats the replacement.

2.3.23 Leaving the Editor

There are several ways to exit the editor and save any changes you may have made to the file. One way is to enter:

:x

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

ZZ

the same thing happens, except the old copy file is written out *only* if you have made any changes. Note that the **ZZ** command is *not* preceded by a colon, and is not echoed on the screen.

To leave the editor without saving any changes you have made to the file, enter:

:q!

The exclamation point tells **vi** to quit unconditionally. If you leave out the exclamation point:

:q

vi will not let you quit. You will see the error message:

No write since last change (:quit! overrides)

This message tells you to use **:q!** if you really want to leave the editor without saving your file.

Saving a File Without Leaving the Editor

There are many occasions when you must save a file without leaving the editor, such as when starting a new shell, or moving to another file. Before you can perform these tasks you must first save the current file with the Write (**:w**) command:

:w

You do not need to enter the name of the file; **vi** remembers the name you used when you invoked the editor. If you invoked **vi** without a filename, you may name the file by entering:

:w *filename*

where *filename* is the name of the new file.

2.3.24 Editing a Series of Files

Entering and leaving **vi** for each new file takes time, particularly on a heavily used system, or when you are editing large files. If you have many files to edit in one session, you can invoke **vi** with more than one filename, and thus edit more than one file without leaving the editor, as in:

vi file1 file2 file3 file4 file5 file6

But entering many filenames is tedious, and you may make a mistake. If you mistype a filename, you must either backspace over to mistake and reenter the line, or kill the whole line and reenter it. It is more convenient to invoke **vi** using the special characters as abbreviations.

To invoke **vi** on the above files without typing each name, enter:

vi file*

This invokes **vi** on all files that begin with the letters "file". You can plan your filenames to save time in later editing. For example, if you are writing a document that consists of many files, it would be wise to give each file the same filename extension, such as ".s". Then you can invoke **vi** on the entire document:

vi *.s

You can also invoke **vi** on a selected range of files:

vi [3-5]*.s

or

vi [a-h]*

To invoke **vi** on all files that are five letters long, and have any extension:

vi ?????.*

For more information on using special characters, see Chapter 3 of the *Introduction to XENIX* manual, section 3.3.4, "Special Characters".

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

x files to edit

After you have finished editing a file, save it with the Write (:w) command, then go to the next file with the Next (:n) command:

:n

The next file appears, ready to edit. It is not necessary to specify a filename; the files are invoked in alphabetical (or numerical, if the filenames begin with numbers) order.

If you forget what files you are editing, enter:

:args

The list of files appears on the status line. The current file is enclosed in square brackets.

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

:e file4

instead of using the (:n) command. If you enter:

:n

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

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

:rew

To discard the changes you made and start again at the beginning, enter:

:rew!

2.3.25 Editing a New File Without Leaving the Editor

You can start editing another file anywhere on the XENIX system without leaving **vi**. This saves time when you wish to edit several files in one session that are in different directories, or even in the same directory. For example, if you have finished editing */usr/joe/memo* and you wish to edit */usr/mary/letter*, first save the file *memo* with the Write (**:w**) command then enter:

:e /usr/mary/letter

/usr/mary/letter appears on your screen just as though you had left **vi**.

Note

You *must* write out your file with the Write (:w) command to save the changes you have made. If you try to edit a second file without writing out the first file, the message "No write since last change (:e! overrides)" appears. If you use **:e!** all your changes to the first file are discarded.

If you want to switch back and forth between two files, **vi** remembers the name of the last file edited. Using the above example, if you wish to go back and edit the file */usr/joe/memo* after you have finished with */usr/mary/letter*, enter:

:e#

The cursor is positioned in the same location it was when you first saved */usr/joe/memo*.

2.3.26 Leaving the Editor Temporarily: Shell Escapes

You can execute any XENIX command from within **vi** using the shell Escape (**!**) command. For example, if you wish to find out the date and time, enter:

:!date

The exclamation point sends the remainder of the line to the shell to be executed, and the date and time appear on the **vi** status line. You can use the **!** to perform any XENIX command. To send mail to joe without leaving the editor, enter:

:!mail joe

Type your message and send it. (For more information about the XENIX mail system, see Chapter 3, "mail".) After you send it, the message

[Hit return to continue]

appears. Press **RETURN** to continue editing.

If you want to perform several XENIX commands before returning to the editor, you can invoke a new shell:

:!sh

The XENIX prompt appears. You may execute as many commands as you like. Press **Ctrl-d** to terminate the new shell and return to your file.

If you have not written out your file before a shell escape, you will see the message:

[No write since last change]

It is a good idea to save your file with the Write (**:w**) command before executing an escape, just in case something goes wrong. However, once you become an experienced **vi** user, you may wish to turn off this message. To turn off the "No write" message, reset the **warn** option, as follows:

:set nowarn

For more information about setting options in **vi**, see section 2.5, "Setting Up Your Environment".

2.3.27 Performing a Series of Line-Oriented Commands: Q

If you have several line-oriented commands to perform, you can place yourself temporarily in Line-oriented mode by entering:

Q

while you are in Command mode. A colon prompt appears on the status line.

Commands executed in this mode cannot be undone with the **u** command, nor do they appear on the screen until you re-enter Normal **vi** mode. To re-enter Normal **vi** mode, enter:

vi

2.3.28 Finding Out What File You're In

If you forget what file you are editing, press **Ctrl-g** while you are in Command mode. A line similar to the following appears appears on the status line:

"memo" [Modified] line 12 of 100 --12%--

From left to right, the following information is displayed:

- The name of the file
- Whether or not the file has been modified
- The line number the cursor is on
- How many lines there are in the file
- Your location in the file (expressed as a percentage)

This command is also useful when you need to know the line number of the current line for a line-oriented command.

The same information can be obtained by entering:

:file

or

:f

2.3.29 Finding Out What Line You're On

To find out what line of the file you are on, enter:

:nu

and press **RETURN**. This command displays the current line number and the text of the line.

To display line numbers for the entire file, see section 2.5.5, "Displaying Line Numbers: number"

2.4 Solving Common Problems

The following is a list of common problems that you may encounter when using **vi**, along with the probable solution.

- *I don't know which mode I'm in.*

  Press **ESC** until the bell rings. When the bell rings you are in Command mode.

- *I can't get out of a subshell.*

  Press **Ctrl-d** to exit any subshell. If you have created more than one subshell (not a good idea, usually), keep pressing **Ctrl-d** until you see the message:

  [Hit return to continue]

- *I made an inadvertent deletion (or insertion).*

  Press **u** to undo the last Delete or Insert command.

- *There are extra characters on my screen.*

  Press **Ctrl-l** to redraw the screen.

- *When I type, nothing happens.*

  vi has crashed and you are now in the shell with your terminal characteristics set incorrectly. To reset the keyboard, slowly enter:

  stty sane

  then press **Ctrl-j** or **LINEFEED**. Pressing **Ctrl-j** instead of **RETURN** is important here, since it is quite possible that the **RETURN** key will not work as a newline character. To make sure that other terminal characteristics have not been altered, logoff, turn your terminal off, turn your terminal back on, and then log back in. This should guarantee that your terminal's characteristics are back to normal. This procedure may vary somewhat depending on the terminal.

- *The system crashed while I was editing.*

  Normally, **vi** will inform you (by sending you mail) that your file has been saved before a crash. The file can be recovered by entering:

  vi -r *filename*

  If **vi** was unable to save the file before the crash, it is irretrievably lost.

- *I keep getting a colon on the status line when I press* **RETURN**

  You are in line-oriented Command mode. Enter:

  vi

  to return to normal **vi** Command mode.

- *I get the error message "Unknown terminal type [Using open mode]" when I invoke vi.*

  Your terminal type is not set correctly. To leave Open mode, press **ESC**, then enter:

  :wq

  and press **RETURN**. Turn to section 2.5.1, "Setting the Terminal Type" for information on how to set your terminal type correctly.

2.5 Setting Up Your Environment

There are a number of options that can be set that affect your terminal type, how files and error messages are displayed on your screen, and how searches are performed. These options can be set with the **set** command while you are editing, or they can be placed in the **vi** startup file, *.exrc*. (The *.exrc* file is explained in section 2.5.13.) The following sections describe the most commonly used options and how to set them. There is a complete list of options in **vi**(C) in the XENIX *Reference Manual*.

2.5.1 Setting the Terminal Type

Before you can use **vi**, you must set the terminal type, if this has not already been done for you, by defining the TERM variable in your *.profile* file. (The *.profile* file is explained in the XENIX *User's Guide.*) The TERM variable is a number that tells the operating system what type of terminal you are using. To determine this number you must find out what type of terminal you are using. Then look up this type in Terminals(M) in the XENIX *Reference Manual.* If you cannot find your terminal type or its number, consult your System Administrator.

For these examples, we will suppose that you are using an HP 2621 terminal. For the HP 2621, the TERM variable is "2621". How you define this variable depends on which shell you are using. You can usually determine which shell you are using by examining the prompt character. The Bourne shell prompts with a dollar sign ($); the C-shell prompts with a percent sign (%).

Setting the TERM variable: The Visual Shell

If you are using the Visual Shell the terminal type has already been set, and you do not need to change it.

Setting the TERM variable: The Bourne Shell

To set your terminal type to 2621 place the following commands in the file *.profile*:

```
TERM=2621
export TERM
```

Setting the TERM variable: The C Shell

To set your terminal type to 2621 for the C shell, place the following command in the file *.login*:

```
setenv TERM 2621
```

2.5.2 Setting Options: The set Command

The **set** command is used to display option settings and to set options.

Listing the Available Options

To get a list of the options available to you and how they are set, enter:

:set all

Your display should look similar to this:

noautoindent	open	noslowopen
autoprint	nooptimize	tabstop=8
noautowrite	paragraphs=IPLPPPQPP LIbp	taglength=0
nobeautify	noprompt	ttytype=h19
directory=/tmp	noreadonly	term=h19
noerrorbells	redraw	noterse
hardtabs=8	report=5	warn
noignorecase	scroll=4	window=8
nolisp	sections=NHSHH HU	wrapscan
nolist	shell=/bin/sh	wrapmargin=0
magic	shiftwidth=8	nowriteany
nonumber	noshowmatch	

This chapter discusses only the most commonly used options. For information about the options not covered in this chapter, see **vi**(C) in the XENIX *Reference Manual*.

Setting an Option

To set an option, use the **set** command. For example, to set the *ignorecase* option so that case is *not* ignored in searches, enter:

set noignorecase

2.5.3 Displaying Tabs and End-of-Line: list

The **list** option causes the "hidden" characters and end-of-line to be displayed. The default setting is **nolist**. To display these characters, enter:

:set list

Your screen is redrawn. The dollar sign ($) represents end-of-line and Ctrl-i (^I) represents the tab character.

2.5.4 Ignoring Case in Search Commands: ignorecase

By default, case is significant in search commands. To disregard case in searches, enter:

:set ignorecase

To change this option, enter:

:set noignorecase

2.5.5 Displaying Line Numbers: number

It is often useful to know the line numbers of a file. To display these numbers, enter:

:set number

This redraws your screen. Numbers appear to the left of the text. To remove line numbers, enter:

:set nonumber

2.5.6 Printing the Number of Lines Changed: report

The **report** option tells you the number of lines modified by a line-oriented command. For example,

:set report=1

reports the number of lines modified, if more than one line is changed. The default setting is:

report=5

which reports the number of lines changed when more than five lines are modified.

2.5.7 Changing the Terminal Type: term

If you are logged in on a terminal that is a different type than the one you normally use, you can check the terminal type setting by entering:

:set term

Press **RETURN**. See section 2.5.1, "Setting the Terminal Type" for more information about TERM variables.

2.5.8 Shortening Error Messages: terse

After you become experienced with **vi**, you may want to shorten your error messages. To change from the default **noterse**, enter:

:set terse

As an example of the effect of **terse**, when **terse** is set the message:

No write since last change, quit! overrides

becomes:

No write

2.5.9 Turning Off Warnings: warn

After you become experienced with **vi**, you may want to turn off the error message that appears if you have not written out your file before a Shell Escape (:!) command. To turn these messages off, enter:

:set nowarn

2.5.10 Permitting Special Characters in Searches: nomagic

The **nomagic** option allows the inclusion of the special characters (. \ $ [ ]) in search patterns without a preceding backslash. This option does *not* affect caret (^) or star (*); they must be preceded by a backslash in searches regardless of **magic**. To set **nomagic**, enter:

:set nomagic

2.5.11 Limiting Searches: wrapscan

By default, searches in **vi** "wrap" around the file until they return to the place they started. To save time you may want to disable this feature. Use the following command:

:set nowrapscan

When this option is set, forward searches go only to the end of the file, and backward searches stop at the beginning.

2.5.12 Turning on Messages: mesg

If someone sends you a message with the **write** command while you are in **vi** the text of the message will appear on your screen. To remove the message from your display you must press **Ctrl-l**. When you invoke **vi**, write permission to your screen is automatically turned off, preventing **write** messages from appearing. If you wish to receive **write** messages while in **vi**, reset this option as follows:

```
:set mesg
```

2.5.13 Customizing Your Environment: The .exrc File

Each time **vi** is invoked, it reads commands from the file named *.exrc* in your home directory. This file sets your preferred options so that they do not need to be set each time you invoke **vi**. A sample *.exrc* file follows:

```
set number
set ignorecase
set nowarn
set report=1
```

Each time you invoke **vi** with the above options, your file is displayed with line numbers, case is ignored in searches, warnings before shell escape commands are turned off, and any command that modifies more than one line will display a message indicating how many lines were changed.

2.6 Summary of Commands

The following tables contain all the basic commands discussed in this chapter.

Entering vi

Typing this:	Does this:
vi *file*	Starts at line 1
vi +n *file*	Starts at line *n*
vi + *file*	Starts last line
vi +/pattern *file*	Starts at *pattern*
vi -r *file*	Recovers *file* after a system crash

Cursor Movement

Pressing this key:	Does this:
h	Moves 1 space left
l	Moves 1 space right
SPACEBAR	Moves 1 space right
w	Moves 1 word right
b	Moves 1 word left
k	Moves 1 line up
j	Moves 1 line down
RETURN	Moves 1 line down
)	Moves to end of sentence
(	Moves to beginning of sentence
}	Moves to beginning of paragraph
{	Moves to end of paragraph
Ctrl-w	Moves to first character of insertion
Ctrl-u	Scrolls up 1/2 screen
Ctrl-d	Scrolls down 1/2 screen
Ctrl-f	Scrolls down one screen
Ctrl-b	Scrolls up one screen

Inserting Text

Pressing	Starts insertion:
i	Before the cursor
I	Before first character on the line
a	After the cursor
A	After last character on the line
o	On next line down
O	On the line above
r	On current character, replaces one character only
R	On current character, replaces until ESC

Delete Commands

Command	Function
dw	Deletes a word
d0	Deletes to beginning of line
d$	Deletes to end of line
3dw	Deletes 3 words
dd	Deletes the current line
5dd	Deletes 5 lines
x	Deletes a character

Change Conunands

Conunand	Function
cw	Changes 1 word
3cw	Changes 3 words
cc	Changes current line
5cc	Changes 5 lines

Search Commands

Command	Function	Example
/and	Finds the next occurrence of *and*	and, stand, grand
?and	Finds the previous occurrence of *and*	and, stand, grand
/^The	Finds next line that starts with *The*	The, Then, There
/[bB]ox/	Finds the next occurrence of *box* or *Box*	
n	Repeats the most recent search, in the same direction	

Search and Replace Commands

Command	Result	Example
:s/pear/peach/g	All *pears* become *peach* on the currentline	
:1,$s/file/directory	Replaces *file* with *directory* from line 1 to the end.	*filename* becomes *directoryname*
:g/one/s//1/g	Replaces every occurrence of *one* with 1.	one becomes 1, oneself becomes 1self, someone becomes some1

Pattern Matching: Special Characters

This character:	Matches:
^	Beginning of a line
$	End of aline
.	Any single character
[]	A range of characters

Leaving vi

Command	Result
:w	Writes out the file
:x	Writes out the file, quits **vi**
:q!	Quits **vi** without saving changes
:!command	Executes *command*
:!sh	Forks a new shell
!!command	Executes *command* and places output on current line
:e *file*	Edits *file* (save current file with :w first)

Options

This option:	Does this:
all	Lists all options
term	Sets terminal type
ignorecase	Ignores case in searches
list	Displays tab and end-of-line characters
number	Displays line numbers
report	Prints number of lines changed by a line-oriented command
terse	Shortens error messages
warn	Turns off "no write" warning before escape
nomagic	Allows inclusion of special characters in search patterns without a preceding backslash
nowrapscan	Prevents searches from wrapping around the end or beginning of a file.
mesg	Permits display of messages sent to your terminal with the **write** command

Replace this Page
with Tab Marked:

mail

Chapter 3

mail

3.1 Introduction

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

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

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

3.2 Demonstration

The **mail** command lets you perform two distinct functions: sending mail and disposing of mail. In this demonstration, we will show you how to send mail to yourself, read a message, delete it, and exit the **mail** program.

3.2.1 Composing and Sending a Message

To begin, enter:

```
mail self
```

where *self* is your user name. Next, enter the following lines. Press **RETURN** at the end of each line.

```
This is a message sent to myself.
I compose a message by entering lines of text.
Press Ctrl-d on a newline to end the message.
```

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

```
~?
```

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

```
~s Sample subject
```

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

```
~c abel
```

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

```
~p
```

This will display the following:

```
Message contains:
To: self
Subject: Sample subject
Cc: abel

This is a message sent to myself.
I compose a message by entering lines of text.
Press Ctrl-d on a newline to end the message.
```

Finally, press **Ctrl-d** by itself on a line, to end the message and send it to those that you have mentioned in the *To:* and the *Cc:* fields. You will exit from the **mail** program and return to the XENIX shell. Once you have sent mail, there is no way to undo the act, so be careful.

3.2.2 Reading mail

Within a short time, you should receive the message:

You have mail.

(You must press **RETURN** before this message will appear on your screen.) This message informs you that the message you have just sent has arrived in your system mailbox. To read this message and any others that may have been sent to you, enter:

mail

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

```
Mail version 3.0 August 30, 1985. Type ? for help.
1 message:
 1 self   Fri Aug 31 12:26 7/188 "Sample subject"
_
```

When there is more than one message in your mailbox, the *most recent* message is displayed at the top of the list. The message at the top of the list has the highest number. The messages are numbered in ascending order from least recent to most recent. The message header includes who sent the message, when it was sent, the number of lines and characters, and the subject of the message. The underscore prompt prompts you to enter a **mail** command. Now enter:

?

to get help on all the available **mail** commands. Next, enter:

p

to see the message that you sent to yourself. **mail** displays the following:

```
>From self Fri Aug 20 12:26:52 1985
To: self
Subject: Sample subject

This is a message sent to myself.
I compose a message by entering lines of text.
Press Ctrl-d on a newline to end the message.
```

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

3.2.3 Leaving mail

If this message has no real use, you can delete it by entering:

```
d
```

To get out of **mail**, enter:

```
q
```

mail then displays the message

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

and returns you to the XENIX shell.

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

3.3 Basic Concepts

It is much easier to use **mail** if you understand the basic concepts that underlie it. The concepts discussed in this section are:

- Mailboxes

- Messages

- Modes
- Command syntax

3.3.1 mailboxes

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

1. *You* decide whether mail is to be placed in the user mailbox; it is not automatically placed there.

2. The user mailbox is *not* the place where mail is initially routed–that place is the system mailbox in the directory */usr/spool/mail.*

3. mail is not picked up *from* your user mailbox.

3.3.2 Messages

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

To:	This field is mandatory. It contains one or more valid user names to which you may send mail.
Subject:	This optional field contains text describing the message.
Cc:	The carbon copy field contains one or more valid names of those who are to receive copies of a message. Message recipients see these names in the received message. This field can be empty.
Bcc:	The blind carbon copy field contains the one or more valid names of people who are to receive copies of a message. Recipients do *not* see these names in the received messages. This field can be empty.

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

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

3.3.3 Modes

Often, the biggest hurdle to using **mail** is understanding what modes of operation are available. This section discusses each mode.

When you invoke **mail** you are using the shell. If you want to mail a letter without entering **mail** command mode, you can do so by entering:

```
mail john < letter
```

Here, the file *letter* is sent to the user *john*.

Note

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

You can enter a message from your shell by entering:

```
mail john
```

Next, enter the text of your message as follows:

```
This is the text of the message.
```

Press RETURN to start a new line, then Ctrl-d to send the message.

Messages such as the one above are created in mail's *compose mode*. When entering text in compose mode, there are several special keys associated with line editing functions: these are the same special characters that are available to you when executing normal XENIX commands. For example, you can kill the line you are editing by entering Ctrl-u, normally the kill character. To backspace, press the BACKSPACE Key or Ctrl-h.

>From compose mode, you can issue commands called compose escapes. These are also called *tilde escapes* because the command letters are preceded by a tilde (~). When you execute these commands you are

temporarily leaving or escaping from compose mode; hence the name. Note that once you have pressed RETURN to end a line, you cannot change that line from within compose mode. You must enter edit mode in order to change that line.

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

mail

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

You can enter *edit mode* from either compose mode or command mode. In edit mode, you edit the body of a message using the full capabilities of an editor. To enter edit mode from command mode, use either the **e** or **edit** command to enter **ed**, or the **v** or **visual** command to enter **vi.** (**Vi** may not be available on your system.) To enter edit mode from compose mode, use the compose escapes ~e and ~v, respectively.

3.3.4 Message-Lists

Many **mail** commands take a list of messages as an argument. A *message-list* is a list of message numbers, ranges, and names, separated by spaces or tabs. Message numbers may be either decimal numbers, which directly specify messages, or one of the special characters ^ , . , or $, *which specify the first, current, or last undeleted* message, respectively. Here, relevant means *not deleted.*

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

p1–4

To display all the messages from the current message to the last message, enter:

p.–$

A *name* is a user name. Messages can be displayed by specifying the name of the sender. For example, to display each message sent to you by *john,* enter:

pjohn

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

    p*

To delete all messages, enter:

    d*

To restore all messages, enter:

    u*

All three of these commands are described later in detail in Section 3.5, "Commands."

3.3.5 Headers

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

- The number of the message
- The sender
- The date sent
- The number of characters and lines
- The subject (if the message contains a *Subject:* field)

Message headers are displayed in *windows* with the **headers** command. A header window contains no more than 18 headers. If there are fewer than 18 messages in the mailbox, all are displayed in one header window. If there are *more* than 18 messages, then the list is divided into an appropriate number of windows. You can move forward one window at a time with the command:

    headers +

and move backward one window at a time with the command:

headers –

commands.

3.3.6 Command Syntax

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

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

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

No applicable messages

3.4 Using mail

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

3.4.1 Entering and Exiting mail

To begin a session with **mail,** enter:

mail

The headers for each received message are then displayed one screenful at a time. To display the next screenful of headers (if any), enter:

```
h+
```

To end the **mail** session, use the **quit** (q) command. All messages remain in the system mailbox unless they have been deleted with the **delete** (d) command, saved with the **save** or **write** command, or held in your user mailbox with the **mbox** command. Deleted messages are discarded. The **−f** command line option causes **mail** to read in the contents of *mbox*. Optionally, a filename may be given as an argument to **−f**, so that the specified file is read instead. When you **quit**, **mail** writes all messages back to this file.

If you send mail over a noisy phone line, you will notice that many of the bad characters turn out to be RUBOUT or DEL character. These characters cause **mail** to abort messages. To deal with this annoyance, you can invoke **mail** with the **−i** option which causes these bad characters to be ignored.

3.4.2 Sending mail

To send a message, invoke **mail** with the names of the people and groups you want to receive the message. Next, enter your message. When you are finished, press Ctrl-d at the beginning of a line. The message is automatically sent to the specified people. While entering the text of your message, you can escape to an editor or perform other useful functions with compose escapes. Section 3.4.5, "Composing mail," describes some features of **mail** available to help you when composing messages.

If you have a file that contains a written message, you can send it to sam, bob, and john by entering:

```
mail sam bob john < letter
```

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

Note

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

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

3.4.3 Reading mail

To read messages sent to you, enter:

    mail

mail then checks your mail out of the system mailbox and prints out a one-line header of each message, one screenful at a time. Enter "h+" to view the next screenful. The most recent message is initially the first message (numbered highest, because messages are numbered chronologically) and may be printed using the **print** command. You can move forward one message by pressing RETURN or entering "+". To move forward *n* messages use "+*n*". You can move backwards one message with the "–" command or move backwards *n* messages and print with "–*n*". You can also move to any arbitrary message and print it by entering its number.

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

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

Enter:

    restart

and the headers of the new messages are displayed.

3.4.4 Disposing of mail

After examining a message you can delete it with the **delete** (d) command, reply to it with the **reply** (r) command, forward it with the **forward** (f) command, or skip to the next message by pressing RETURN. Deletion causes the mail program to forget about the message. This is not irreversible; the message can be *undeleted* with the **undelete (u) command by entering:**

u *number*

3.4.5 Composing mail

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

```
mail john
```

where john is the name of a user to whom you want to send mail. >From **mail** command mode, you can enter compose mode with the **mail, reply,** or **Reply** commands. Once in compose mode, the text that you enter is appended one line at a time to the body of the message you are sending. Normal line editing functions are available when entering text, including Ctrl-u to kill a line and Backspace to back up one character. Note that when you enter two interrupts in a row (i.e., pressing INTERRUPT twice), your composition is aborted.

While you are composing a message, **mail** treats lines beginning with the tilde character (~) in a special way. This character introduces commands called compose escapes. For example, entering:

```
~m
```

by itself on a line places a copy of the most recently printed message inside the message you are composing. The copy is shifted right one tabstop.

Other escapes set up heading fields, add and delete recipients to the message, allow you to escape to an editor, let you revise the message body, or run XENIX commands. To get a list of the available compose escapes when in compose mode, enter:

```
~?
```

See also Section 3.6, "Leaving Compose Mode Temporarily," later in this chapter.

3.4.6 Forwarding mail

To forward a message, use the **forward** (f) command. For example, enter:

```
f john
```

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

3.4.7 Replying to mail

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

3.4.8 Specifying Messages

Commands such as **print** and **delete** can be given a message-list argument to apply to several messages at once. Thus "delete 2 3" deletes messages 2 and 3, while "delete 1–5" deletes messages 1 through 5. A star (*) addresses all messages, and a dollar sign ($) addresses the last (highest numbered) message. The **top** (t) command displays the first five lines of a message; hence, you can enter:

```
top *
```

to display the first five lines of every message. Message-lists can contain combinations of lists, ranges, and names. For example, the following command displays all messages from tom or bob and numbered 2, 4, 10, 11, or 12:

```
p tom bob 2 4 10-12
```

3.4.9 Creating Mailing Lists

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

```
alias cohorts bill bob barry
```

in the file *.mailrc* in your home directory. The current list of such aliases can be displayed with the **alias (a) mail** command. Personal aliases are expanded in mail sent to others so that they will be able to **Reply** to each individual recipient. For example, the *To:* field in a message sent to *cohorts* will read:

```
To: bill bob barry
```

and not:

```
To: cohorts
```

Normally, system-wide aliases are available to all users. These are installed by whoever is in charge of your system. For more information, see section 3.8, "Using Advanced Features," later in this chapter.

3.4.10 SendingNetwork 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 your system, you can send mail to sites within the UUCPnetwork 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 *Operations Guide*. For more information about UUCP, see the XENIX *Reference Manual*.

3.4.11 Setting Options

mail has several options that you can **set** from **mail** command mode or in the file *.mailrc* in your home directory. For example, "set askcc" enables the **askcc** switch and causes prompting for additions to the *Cc:* field when you finish composing a message. These and other options are discussed in Section 3.7 "Setting Up Your Environment: The .mailrc File."

3.5 Commands

This section describes each of the commands available to you in **mail** command mode. The examples in this section assume you have invoked **mail** and that you have several messages you want to dispose of. Note that in general, **mail** commands can be invoked with either the name of the command or a one- or two-character mnemonic abbreviation. In the text of the command descriptions below, this mnemonic abbreviation is enclosed in parentheses after the name of the command. All commands are printed in boldface, except in the examples.

3.5.1 Getting Help: help and ?

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

?

or:

help

3.5.2 Reading mail: p, +, –, and *restart*

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

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

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

To examine the second message, enter:

p2

This might cause **mail** to respond with:

```
Message 2:
>From sam Tue June 20 22:55 1985
Subject: Meeting

Meeting everyone, please do not forget!
```

To look at message 3, enter:

or to look at message 1, enter:

+

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

+3

skips forward three messages. If you enter:

p *

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

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

.

To display message 1, enter:

^

To display message 3, enter:

$

When new mail arrives while you are in **mail**, the message "New mail has arrived–type 'restart' to read." If you wish to read the new messages, enter:

restart

The headers of the new messages appear.

3.5.3 Finding Out the Number of the Current Message: =

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

3.5.4 Displaying the First Five Lines : t

The **top** (t) command takes a message-list and displays the first five lines of each addressed message. For example:

top2-12

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

3.5.5 Displaying Headers: h

The **headers** (h) command displays header windows or lists of headers. A header window contains no more than 18 headers. With no argument, the **headers** command displays a header window in which the current message header is displayed at the center of the window.

To examine the next set of 18 headers, enter:

h +

To examine the previous set, enter:

h –

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

h joe

displays all the message headers from joe. The following are some characteristics of the header-list:

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

3.5.6 Deleting Messages: d and dp

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

delete 1

prevents **mail** from retaining message 1 in the system mailbox. The message will disappear altogether, along with its number.

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

3.5.7 Undeleting Messages: u

The **undelete** (u) command causes a message that has been previously deleted with **d** or **dp** to reappear as if it had never been deleted. For example, to undelete message 3, enter:

```
u3
```

You cannot undelete messages from previous **mail** sessions; they are permanently deleted.

3.5.8 Leaving mail : q and x

When you have read all your messages, you can leave **mail** with the **quit** (q) command. All messages are held in your system mailbox, except the following:

- Deleted messages, which are discarded irretrievably.
- Messages marked with the **mbox** command, which are saved in *mbox* in your home directory (that is, your user mailbox).
- Messages saved with the **save** and **write** commands are deleted from the system mailbox. Forwarded messages are *not* deleted.

Note that if the *autombox* option is set, messages that you have read are automatically saved in your user mailbox. If you wish to leave **mail** quickly without altering either your system or user mailbox, you can use the **exit** (x) command. This returns you to the shell without changing anything: no messages are deleted or saved. Files that you invoke with the **mail -f** switch are unaffected as well.

3.5.9 Saving Your mail: s

The **save** (s) command lets you save messages to files other than *mbox*. By using **save**, you can organize your mail by putting messages in appropriate files. The **save** command writes out each message to the file given as the last argument on the command line. For example, the following command appends messages 1-5 to the file *letters* :

```
s 1-5 letters
```

The file *letters* is created if it does not already exist. Saved messages are not automatically retained in the system mailbox when you quit, nor are they selected by the **print** command described above, unless explicitly requested. Each saved message is marked with a star (*).

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

3.5.10 Saving Yourmail: w

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

```
w3-17john elliot book
```

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

3.5.11 SavingYourmail:mb

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

3.5.12 Saving Yourmail: ho

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

3.5.13 Printing Your mail on the Lineprinter: l

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

```
l doug
```

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

3.5.14 Sending mail: m

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

3.5.15 Replying to mail: r and R

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

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

3.5.16 Forwarding mail: f and F

To forward a copy of a message, use the **forward** (f) command. This causes a copy of the current message to be sent to the specified users. The message is marked as saved, and then deleted from the system mailbox when you exit **mail.** For example, to forward the current message to someone whose login name is john, enter:

```
f john
```

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

```
f 2 john bill
```

forwards message 2 to john and bill.

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

3.5.17 Creating mailing Lists: a

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

alias beatles john paul george ringo

so that whenever you used the name *beatles* in a destination address (as in "mail beatles"), it would be expanded so that you are really referring to the four names aliased to *beatles.* With no arguments, **alias** displays all currently-defined aliases. With one argument, it prints out the users defined by the given alias.

You will probably want to define aliases in the startup file, *.mailrc,* so that you do not have to redefine them each time you invoke **mail**. See section 6.7, "Setting Up Your Environment: The .mailrc File," for more information.

3.5.18 Setting and Unsetting Options: se and uns

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

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

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

The command

set ?

displays a list of the available options. See the section "Setting Up Your Environment," for descriptions of these options.

3.5.19 Editing a Message: e and v

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

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

3.5.20 Executing Shell Commands: sh and !

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

    !date

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

    sh

To exit from this new shell and return to **mail** command mode, press Ctrl-d.

3.5.21 Finding Out the Number of Characters in a Message: si

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

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

3.5.22 Changing the Working Directory: cd

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

3.5.23 Reading Commands From a File: so

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

3.6 Leaving Compose Mode Temporarily

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

3.6.1 Getting Help: ~?

The help escape is the first compose escape you should know because it tells you about all the others. For example, if you enter:

```
~?
```

a brief summary of the available compose escapes is displayed on your screen. Note that ~h prompts for heading fields and and does *not* give help.

3.6.2 Printing the Message: ~p

To print the current text of a message you are composing, enter:

```
~p
```

This prints a line of dashes and the heading and body of the message so far.

3.6.3 Editing the Message: ~e and ~v

If you are dissatisfied with a message as it stands, you can edit the message by invoking the editor, **ed**, with the editor escape, ~e. This causes the message to be copied into a temporary file so that you can edit it. Similarly, the ~v escape causes the message to be copied into a temporary file so that you can edit it with the **vi** editor. After modifying the message to your satisfaction, write it out and quit the editor. **mail** responds by entering:

(continue)

after which you may continue composing your message.

3.6.4 Editing Headers: ~t, ~c, ~b, ~s, ~R and ~h

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

~t name1 name2 ...

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

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

~s *line-of-text*

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

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

~c name1 name2 ...

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

~cc name1 name2 ...

performs an identical function. Similarly, the escape:

~b name1 name2 ...

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

The escape:

~R

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

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

~h

where h stands for "heading." The escape **~h** displays *To:* followed by the current list of recipients and leaves the cursor at the end of the line. If you enter ordinary characters, they are appended to the end of the current list of recipients. You can also use the normal XENIX command line editing characters to edit these fields, so you can erase existing heading text by backspacing over it.

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

3.6.5 Adding a File to the Message: ~r and ~d

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

~r filename

is provided for this purpose, and causes the named file to be appended to your current message. **mail** complains if the file does not exist or cannot be read. If the read is successful, **mail** displays the number of lines and characters appended to your message.

As a special case of ~r, the escape:

~d

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

3.6.6 Enclosing Another Message: ~m and ~M

If you are sending mail from within mail's command mode, you can insert a message, that was previously sent to you, into the message that you are currently composing. For example, you might enter:

~m 4

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

~M 4

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

3.6.7 Saving the Message in a File: ~w

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

~w filename

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

3.6.8 Leaving mail Temporarily: ~! and ~|

To temporarily escape to the shell, use the escape:

~!*command*

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

~|*command*

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

(continue)

3.6.9 Escaping to mail Command Mode: ~:

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

> ~:*mail-command*
>
> ~_*mail-command*

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

> May not execute *cmd* while composing

3.6.10 Placing a Tilde at the Beginning of a Line: ~~

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

> ~~This line begins with a tilde.

appends:

> ~This line begins with a tilde.

to your message. The escape character can be changed to a different character with the *escape* option. (For information on how to set options, see section 6.7, "Setting Up Your Environment: The .mailrc File.") If the escape character is not a tilde, then this discussion applies to that character and not the tilde.

3.7 Setting Up Your Environment: The .mailrc File

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

```
# number sign introduces comments

# personal aliases office and cohorts are defined below

alias office bill steve karen
alias cohorts john mary bob beth mike

# set dot lets messages be terminated by period on new line

# set askcc says to prompt for Cc: list after composing message

set dot askcc

# cd changes directory to different current directory

cd
```

3.7.1 The Subject Prompt: asksubject

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

3.7.2 The CC Prompt: askcc

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

```
interrupt
(continue)
```

so that you can return to editing your message.

3.7.3 Printing the Next Message: autoprint

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

3.7.4 Listing Messages in Chronological Order: chron and mchron

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

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

3.7.5 Using the Period to Send a Message: dot

The *dot* switch lets you use a period (.) as an end-of-transmission character, as well as Ctrl-d. This option is available for those who are used to this convention when editing with the editor, *ed*.

3.7.6 Sending mail While in mail: execmail

It is often desirable to reply to a piece of mail, or send mail while reading your mail file. This process is speeded up by the use of the *execmail* option. It causes the underbar prompt to return before **mail** is finished being sent. This frees the user to continue while **mail** performs mailing functions in the background.

3.7.7 Including Yourself in a Group: metoo

Usually, when a group is expanded that contains the name of the sender, the sender is removed from the expansion. Setting the **metoo** option causes the sender to be included in the group.

3.7.8 Saving Aborted Messages: save

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

3.7.9 Printing the Version Header: quiet

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

3.7.10 Choosing an Editor: The EDITOR String

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

    set EDITOR=/bin/ed

3.7.11 Choosing an Editor: The VISUAL String

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

    set VISUAL=/bin/vi

By default, vi is the editor used.

3.7.12 Choosing a Shell: The SHELL String

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

    set SHELL=/bin/sh

3.7.13 Changing the Escape Character: The escape String

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

    set escape=*

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

3.7.14 Setting Page Size: The page String

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

3.7.15 Saving Outgoing mail: The record String

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

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

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

3.7.16 Keeping mail in the System mailbox: autombox

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

3.7.17 Changing the top Value: The toplines String

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

```
set toplines=10
```

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

3.7.18 Sending mail Over Telephone Lines: ignore

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

3.8 Using Advanced Features

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

3.8.1 Command Line Options

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

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

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

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

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

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

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

```
mail -r "meeting" -b singleuser -c "xyz" user user2
```

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

```
mail -f filename
```

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

```
mail -f
```

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

If you send mail over a noisy phone line, you may notice that bad characters are transmitted. These are characters that abort messages: RUBOUT and DEL. You can invoke **mail** with the **-i** switch to ignore these bad characters.

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

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

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

3.8.2 Using mail as a Reminder Service

Besides sending and receiving mail, you can use **mail** as a reminder service. Several XENIX commands have this idea built in to them. For example, the XENIX **lp** command's **-m** switch causes mail to be sent to the user after files have been printed on the lineprinter. XENIX automatically examines the file named *calendar* in each user's home directory and looks for lines containing either today or tomorrow's date. These lines are sent by **mail** as a reminder of important events.

If you program in the shell command language, you can use **mail** to signal the completion of a job. For example, you might place the following two lines in a shell procedure:

```
biglongjob
echo "biglongjob done" |mail self
```

You can also create a a logfile that you want to mail to yourself. For example, you might have a shell procedure that looks like this:

```
dosomething >logfile
mail self <logfile
```

For information about writing shell procedures, see Chapter 4 of this manual, "The Shell."

3.8.3 Handling Large Amounts of mail

Eventually, you will face the problem of dealing with an accumulation of messages in your user mailbox. There are a number of strategies that you can employ to solve this problem concerning space in your mailbox file. Keep in mind the dictum:

When in doubt, throw it out.

This means that you should only save *important* mail in your user mailbox. If your mailbox file becomes large, you must periodically examine its contents to decide whether messages are still relevant. To save space, consider summarizing very long messages.

The previously mentioned measures are not always helpful enough in organizing the many messages that you are likely to receive. Another effective approach is to save mail in files organized by sender, by topic, or by a combination of the two. Create these files in a separate **mail** directory; you can access these mailbox files with the **mail** -f *filename* switch. However, be forewarned–this approach to organizing mail quickly eats up disk space.

3.8.4 Maintenance and Administration

The following is a list of the programs and files that make up the XENIX **mail** system:

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

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

3.9 Quick Reference

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

3.9.1 Command Summary

Given below are the name and syntax for each command, the abbreviated form (in brackets), and a short description. Many commands have optional arguments; most can be executed without any arguments at all. In particular, commands that take a message-list argument will default to the current message if no message-list is given. In the following descriptions, boldface denotes the name of a command, compose escape or option. Italics are used for arguments to commands or compose escapes. The vertical bar indicates selection and is used to separate the arguments from which you may select. All other text should be read literally.

RETURN	Displays the next message.
+*n*	[**+**] With no *n* argument, it displays the next message. If given a numeric argument *n*, goes to the *n*th message and displays it.
−*n*	[**−**] With no *n* argument, goes to the previous message and displays it. If given a numeric argument *n*, goes to the *n*th previous message and displays it.
^	Displays the first message.
$	Displays the last message.
=	Displays the message number of the current message.
?	Displays the summary of **mail** commands in */usr/lib/mail/mailhelp.cmd.*
!*shell-cmd*	Executes the shell command that follows. No space is needed after the exclamation point.
Alias *users*	Displays system-wide aliases for users. At least one user must be specified.
alias *name users*	[**a**] Aliases *users* to *name*. With no name arguments, displays all currently defined aliases. With one argument, displays the users aliased by the given name argument.
cd *directory*	[**c**] Changes the user's working directory to the specified directory. If no directory is given, then changes to the user's home directory.
delete *mesg-list*	[**d**] Deletes each message in the given message-list.

dp *mesg-list* — Deletes the current message and displays the next message.

echo *path* — Expands shell metacharacters.

edit *mesg-list* — [e] Takes the given message-list and points the text editor at each message in turn. On return to command mode, the edited message is read back in. See also the **visual** command.

exit[!] — [x] Immediately returns to the shell without modifying the system mailbox, the user mailbox, or a file specified with the **−f** switch.

file — [fi] Displays the name of the mailbox file.

forward *mesg-num user-list*
[f] Takes a *user-list* argument and forwards the current message to each name. The message sent to each is indented and shows that the sender has passed it on. The *mesg-num* argument is optional, and is used to forward the numbered message instead of the default message.

Forward *mesg-num user-list*
[F] Same as **forward** except that the message is not indented.

headers *+n* |*-n* |*mesg-list*
[h] With no argument, lists the current range of headers, which is an 18-message group. If a plus (+) argument is given, then the next 18-message group is displayed, and if a minus (−) argument is given, the previous 18-message group is displayed. Both plus and minus accept an optional numeric argument indicating the number of header-windows to move forward or backward. If a message-list is given, then the message-header for each message in the list is displayed.

help — Same as ? above. Prints the summary of **mail** commands in */usr/lib/mail/mailhelp.cmd.*

hold *mesg-list* — [ho] Takes a message-list and marks each message to be saved in the user's system mailbox instead of in *mbox.*

list Prints list of **mail** commands.

lpr *mesg-list* [l] Prints each of the messages in the required message-list on the lineprinter. Messages are piped through *pr* before being printed.

mail *[user-list]* [m] Takes an optional user-list argument and sends mail to each name after entering compose mode.

mbox *mesg-list* [mb] Marks messages given in the message-list argument to be saved in the user mailbox when a **quit** is executed. Message headers contain an initial letter "M" to show that they are to be saved.

move *mesg-list mesg-num*
Places the messages specified in *mesg-list* after the message specified in *mesg-num*. If *mesg-num* is 0, *mesg-list* moves to the top of the mailbox.

print *mesg-list* [p] Takes a message-list and displays each message on the user's terminal.

quit [q] Terminates the **mail** session, retaining all nondeleted, unsaved messages in the system mailbox. If the **autombox** option is set, then examined messages are saved in the user mailbox, deleted messages are discarded, and all messages marked with the **hold** command are retained in the system mailbox.

If you are executing **a quit** while editing a mailbox file with the **-f** flag, the mailbox file is rewritten and the user returns to the shell.

reply *mesg-list* [r] Takes a message-list and sends mail to each message author just like the **mail** command.

Reply *mesg-list* [R] Sends a reply to users named in the *To:* and *Cc:* fields, as well as the original sender.

restart Reads in mail that arrives during the current mail session.

save *mesg-list filename*
[s] Takes an optional message-list and a filename and appends each message in turn to

	the end of the file. The default message is the current message.
set	[**se**] Displays a list of available options.
set *option-list*	[**se**] With no arguments, displays all variable values. Otherwise, sets option. Arguments are of the form *option=value,* if the option is a string option or just *option,* if the option is a switch. Multiple options maybe seton one line.
shell	[**sh**] Invokes an interactive version of the shell.
size *mesg-list*	[**si**] Takes a message-list and displays the size in characters of each message.
source *file*	[**so**] Reads and executes **mail** commands from the named file.
string *string mesg-list*	Searches for *string* in *mesg-list.* If no *mesg-list* is specified, all undeleted messages are searched. Ignores case in search.
top	[**t**] Takes a message-list and displays the top five lines. The number of lines displayed is set by the variable *toplines.*
undelete *mesg-list*	[**u**] Takes a message-list and marks each one as not being deleted. Each message in the list must previously have been deleted.
unset *options*	[**uns**] Takes a list of option names and discards their remembered values; this is the opposite of **set**.
visual *mesg-list*	[**v**] Takes a message-list and invokes the **vi** editor on each one.
whois	Looks up a list of target mail recipients and prints the real names or descriptions of each recipient. If the first character of the first argument is alphabetic, the arguments are looked up without change. Otherwise, the arguments are assumed to be a message list, in the format specified in the *mail User's Guide.* For each message in the list, the "From" person is extracted from the header and added to list of users to be searched.

write *mesg-list filename*
[w] Writes the message bodies of messages given by the message-list to the file given by *filename*.

3.9.2 Compose Escape Summary

Compose escapes are used when composing messages to perform special functions. They are only recognized at the beginning of lines. The escape character can be set with the *escape* string option. (See section 6.7.14, "The escape String.") Abbreviations for each escape are in brackets.

Here is a summary of the compose escapes:

~~*string* — Inserts the string of text in the message prefaced by a single tilde (~).

~? — Prints out help for compose escapes on terminal.

~. — Same as Ctrl-d on a newline.

~!*command* — Executes a shell command, then returns to compose mode.

~|*command* — Pipes the message body through the command as a filter. Replaces the message body with the output of the filter. If the command gives no output or terminates abnormally, retains the original message body.

~_*mail-command* — Executes a **mail** command, then returns to compose mode.

~:*mail-command* — Executes a **mail** command, then returns to compose mode.

~**alias** — [~a] Displays a list of private aliases.

~**alias** *aliasname* — [~a] Displays the names included in private *aliasname*.

~**alias** *aliasname users*
[~a] Adds *users* to private *aliasname* list.

~**Alias** — [~A] Performs aliasing by first examining private aliases and then system-wide aliases using all three global alias files. Only the final result is printed (non-local mail recipients will have the complete delivery path printed). The user list is taken from header fields.

~Alias *users* — [~**A**] Performs aliasing by first examining private aliases and then system-wide aliases using all three global alias files. Only the final result is printed (non-local mail recipients will have the complete delivery path printed). At least one user must be specified.

~bcc *name ...* — [~**b**] Adds the given names to the *Bcc:* field.

~cc *name ...* — [~**c**] Adds the given name to the *cc:* field.

~dead — [~**d**] Reads the file *dead.letter* from your home directory into the message.

~editor — [~**e**] Invokes the line editor on the message being sent. Exiting the editor returns the user to compose mode.

~headers — [~**h**] Edits the message heading fields by printing each one in turn and allowing the user to modify each field.

~message *mesg-list* — [~m] Reads the named messages into the message being sent, shifted right one tab. If no messages are specified, reads the current message.

~Message *mesg-list* — [~**M**] Same as ~message except with no right shift.

~print — [~**p**] Prints the message buffer prefaced by the message heading.

~Print — [~**P**] Prints the real names or descriptions (in parentheses) after each recipient.

~quit — [~**q**] Aborts the message being sent, copying the message to *dead.letter* in your home directory if the *save* option is set.

~read *filename* — [~**r**] Reads the named file into the message.

~Return *name* — [~**R**] Adds the given names to the *Return-receipt-to:* field.

~shell — [~**sh**] Invokes a shell.

~subject *string* — [~s] Causes the named string to become the current subject field.

~to *name* **...**	[**~t**] Adds the given names to the *To:* field.
~visual	[**~v**] Invokes the **vi** editor to edit the message buffer. Exiting the editor returns the user to compose mode.
~write *filename*	[**~w**] Writes the message body to the named file.

3.9.3 Option Summary

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

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

mchron Causes messages to be listed in numerical order (most recently received first), but displayed in chronological order.

metoo Normally, before sending, the name of the sender is removed from alias expansions. If *metoo* is set, then the name of the sender is *not* removed.

nosave Prevents saving of the message buffer in the file *dead.letter* in the home directory, after two consecutive interrupts or a ~q escape.

page=*n* Specifies the number of lines (*n*) to be printed in a "page" of text when displaying messages.

quiet Suppresses the printing of the version when **mail** is first invoked.

record= Sets the pathname of the file used to record all outgoing mail. If not defined, then outgoing mail is *not* copied.

SHELL= Pathname of the shell to use in the ! command and the ~! escape. A default shell is used if this option is not defined.

toplines= Sets the number of lines of a message to be printed with the **top** command. Default is five lines.

verify Causes each target mail recipient to be verified. This option permits errors made while composing messages to be corrected or ignored.

VISUAL= Pathname of the text editor to use in the **visual** command and ~v escape. The default is for the **vi** editor.

Replace this Page
with Tab Marked:

Shell

Chapter 4

The Shell

4.1 Introduction

When users log into XENIX , they communicate with one of serveral interpreters. This chapter discusses the shell command interpreter, **sh**. This interpreter is a XENIX program that supports a very powerful command language. Each invocation of this interpreter is called a shell; and each shell has one function: to read and execute commands from its standard input.

Because the shell gives the user a high-level language in which to communicate with the operating system, XENIX can perform tasks unheard of in less sophisticated operating systems. Commands that would normally have to be written in a traditional programming language can be written with just a few lines in a shell procedure. In other operating systems, commands are executed in strict sequence. With XENIX and the shell, commands can be:

- Combined to form new commands
- Passed positional parameters
- Added or renamed by the user
- Executed within loops or executed conditionally
- Created for local execution without fear of name conflict with other user commands
- Executed in the background without interrupting a session at a terminal

Furthermore, commands can "redirect" command input from one source to another and redirect command output to a file, terminal, printer, or to another command. This provides flexibility in tailoring a task for a particular purpose.

4.2 Basic Concepts

The shell itself (that is, the program that reads your commands when you login 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.

4.2.1 How Shells Are Created

In XENIX , a process is an executing entity complete with instructions, data, input, and output. All processes have lives of their own, and may even start (or "fork") new processes. Thus, at any given moment several processes may be executing, some of which are "children" of other processes.

Users log into the operating system and are assigned a "shell" from which they execute. This shell is a personal copy of the shell command interpreter that is reading commands from the keyboard: in this context, the shell is simply another process.

In the XENIX multitasking environment, files may be created in one phase and then sent off to be processed in the "background." This allows the user to continue working while programs are running.

4.2.2 Commands

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

```
lpr allan barry calvin
```

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

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

4.2.3 How the Shell Finds Commands

The shell normally searches for commands in three distinct locations in the file system. The shell attempts to use the command name as given; if this fails, it prepends the string */bin* to the name. If the latter is unsuccessful, it prepends */usr/bin* to the command name. The effect is to search, in order, the current directory, then the directory */bin*, and finally, */usr/bin*. For example, the **pr** and **man** commands are actually the files */bin/pr* and */usr/bin/man*, respectively. A more complex pathname may be given, either to locate a file relative to the user's current directory, or to access a command with an absolute pathname. If a given command name includes a slash (/) (for example, */bin/sort dir/cmd)*, the prepending is not performed. Instead, a single attempt is made to execute the command as named.

This mechanism gives the user a convenient way to execute public commands and commands in or near the current directory, as well as the ability to execute any accessible command, regardless of its location in the file structure. Because the current directory is usually searched first, anyone can possess a private version of a public command without affecting other users. Similarly, the creation of a new public command does not affect a user who already has a private command with the same name. The particular sequence of directories searched may be changed by resetting the shell PATH variable. (Shell variables are discussed later in this chapter.)

4.2.4 Generation of Argument Lists

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

Most characters in such a pattern match themselves, but there are also XENIX special characters that may be included in a pattern. These special characters are: the star (*), which matches any string, including the null string; the question mark (?), which matches any one character; and any sequence of characters enclosed within brackets ([and]), which matches any one of the enclosed characters. Inside brackets, a pair of characters separated by a dash (–) matches any character within the range of that pair.

Thus [a–de] is equivalent to [abcde]. Examples of metacharacter usage:

*	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 that are named in a structured fashion, using common characters or extensions to identify related files.

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

Note that directory names should not contain any of the following characters:

```
* ? [ ]
```

If these characters are used, then infinite recursion may occur during pattern matching attempts.

4.2.5 Quoting Mechanisms

Several characters, including <,>,*,?,[and], have special meanings to the shell. To remove the special meaning of these characters requires some form of quoting. This is done by using single quotation marks (') or double quotation marks (")to surround a string. A backslash (\) before a single character provides this function. (Back quotation marks (`) are used only for command substitution in the shell and do not hide the special meanings of any characters.)

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

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

results in the string:

```
echo $?$*;ls * | wc
```

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

Within double quotation marks, the special meaning of certain characters does persist, while all other characters are taken literally. The characters that retain their special meaning are the dollar sign ($), the backslash (\), the back quotation mark (`), and the double quotation mark (") itself. Thus, within double quotation marks, variables are expanded and command substitution takes place (both topics are discussed in later sections). However, any commands in a command substitution are unaffected by double quotation marks, so that characters such as star (*) retain their special meaning.

To hide the special meaning of the dollar sign ($) and single and double quotation marks within double quotation marks, precede these characters with a backslash (\). Outside of double quotation marks, preceding a character with a backslash is equivalent to placing single quotation marks around that character. A backslash (\) followed by a newline causes that newline to be ignored. The backslash-newline pair is therefore useful in allowing continuation of long command lines.

Some examples of quoting are displayed below:

Input	Shell interprets as:
'`'	The back quotation mark (`)
'"'	The double quotation mark (")
'`echo one`'	the one word "`echo one`"
"\""	The double quotation mark (")
"`echo one`"	the one word "one"
"`"	illegal (expects another `)
one two	the two words "one" & "two"
"one two"	the one word "one two"
'one two'	the one word "one two"
'one * two'	the one word "one * two"
"one * two"	the one word "one * two"
`echo one`	the one word "one"

4.2.6 Standard Input and Output

In general, most commands do not know or care whether their input or output is coming from or going to a terminal or a file. Thus, a command can be used conveniently either at a terminal or in a pipeline. A few commands vary their actions depending on the nature of their input or output, either for efficiency, or to avoid useless actions (such as attempting random access I/O on a terminal or a pipe).

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

An argument to the shell of the form "<*file*" or ">*file*" opens the specified file as the standard input or output (in the case of output, destroying the previous contents of *file*, if any). An argument of the form ">>*file*" directs the standard output to the end of *file*, thus providing a way to append data to the file without destroying its existing contents. In either of the two output cases, the shell creates *file* if it does not already exist. Thus:

```
> output
```

alone on a line creates a zero-length file. The following appends to file *log* the list of users who are currently logged on:

```
who >> log
```

Such redirection arguments are only subject to variable and command substitution; neither blank interpretation nor pattern matching of filenames occurs after these substitutions. This means that:

```
echo 'this is a test' > *.gal
```

produces a one-line file named **.gal*. Similarly, an error message is produced by the following command, unless you have a file with the name "?":

```
cat < ?
```

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

4.2.7 Diagnostic and Other Outputs

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

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

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

This method may be generalized to allow redirection of output associated with any of the first ten file descriptors (numbered 0-9). For instance, if *cmd* puts output on file descriptor 9, then the following line will direct that output to the file *savedata*:

```
cmd 9> savedata
```

A command often generates standard output and error output, and might even have some other output, perhaps a data file. In this case, one can redirect independently all the different outputs. Suppose, for example, that *cmd* directs its standard output to file descriptor 1, its error output to file descriptor 2, and builds a data file on file descriptor 9. The following would direct each of these three outputs to a different file:

```
cmd >standard  2> error  9> data
```

4.2.8 Command Lines and Pipelines

A sequence of commands separated by the vertical bar (|) makes up a *pipeline*. In a pipeline consisting of more than one command, each command is run as a separate process connected to its neighbors by *pipes*, that is, the output of each command (except the last one) becomes the input of the next command in line.

A *filter* is a command that reads its standard input, transforms it in some way, then writes it as its standard output. A pipeline normally consists of a series of filters. Although the processes in a pipeline are permitted to execute in parallel, each program needs to read the output of its predecessor. Many commands operate on individual lines of text, reading a line, processing it, writing it out, and looping back for more input. Some must read large amounts of data before producing output; **sort** is an example of the extreme case that requires all input to be read before any output is produced. The following is an example of a typical pipeline:

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

nroff is a text formatter available in the XENIX Text Processing System whose output may contain reverse line motions, **col** converts these motions to a form that can be printed on a terminal lacking reverse-motion capability, and **lpr** does the actual printing. The flag −mm indicates one of the commonly used formatting options, and *text* is the name of the file to be formatted.

The following examples illustrate the variety of effects that can be obtained by combining a few commands in the ways described above. It may be helpful to try these at a terminal:

- who
  Prints the list of logged-in users on the terminal screen.

- who >>log
  Appends the list of logged-in users to the end of file *log*.

- who | wc −l
  Prints the number of logged-in users. (The argument to wc is pronounced "minus ell".)

- who | pr
  Prints a paginated list of logged-in users.

- who | sort
  Prints an alphabetized list of logged-in users.

- who | grep bob
  Prints the list of logged-in users whose login names contain the string *bob*.

- who | grep bob | sort | pr
  Prints an alphabetized, paginated list of logged-in users whose login names contain the string *bob*.

- { date;who | wc −l; } >> log
  Appends (to file *log*) the current date followed by the count of logged-in users. Be sure to place a space after the left brace and a semicolon before the right brace.

- who |sed −e 's/ .*//' | sort | uniq −d
  Prints only the login names of all users who are logged in more than once. Note the use of **sed** as a filter to remove characters trailing the login name from each line. (The ".*" in the **sed** command is preceded by a space.)

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

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

is the same as:

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

4.2.9 Command Substitution

Any command line can be placed within back quotation marks (\`. . .\`) so that the output of the command replaces the quoted command line itself. This concept is known as *command substitution*. The command or commands enclosed between back quotation marks are first executed by the shell and then their output replaces the whole expression, back quotation marks and all. This feature is often used to assign to shell variables. (Shell variables are described in the next section.)

For example:

```
today=`date`
```

assigns the string representing the current date to the variable "today"; for example "Tue Nov 26 16:01:09 EST 1985". The following command saves the number of logged-in users in the shell variable *users* :

```
users=`who | wc -l`
```

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

```
logmsg=`echo Your login directory is \`pwd\``
```

will display the line "your login directory is *name of login directory*". Shell variables can also be given values indirectly by using the **read** and **line** commands. The **read** command takes a line from the standard input (usually your terminal) and assigns consecutive words on that line to any variables named.

For example:

```
read first init last
```

takes an input line of the form:

```
G. A. Snyder
```

and has the same effect as entering:

```
first=G. init=A. last=Snyder
```

The **read** command assigns any excess "words" to the last variable.

The **line** command reads a line of input from the standard input and then echoes it to the standard output.

4.3 Shell Variables

The shell has several mechanisms for creating variables. A variable is a name representing a string value. Certain variables are referred to as *positional parameters*; these are the variables that are normally set only on the command line. Other shell variables are simply names to which the user or the shell itself may assign string values.

4.3.1 Positional Parameters

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

```
while test -n "$1"
      do case $1 in
            -a) A=aoption ; shift ;;
            -b) B=boption ; shift ;;
            -c) C=coption ; shift ;;
      -*) echo "bad option" ; exit 1 ;;
      *) process rest of files
      esac
done
```

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

```
set abc def ghi
```

assigns the string "abc" to the first positional parameter, $1, the string "def" to $2, and the string "ghi" to $3. Note that $0 may not be assigned a value in this way– it always refers to the name of the shell procedure; or in the login shell, to the name of the shell.

4.3.2 User-Defined Variables

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

name=string

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

More than one assignment may appear in an assignment statement, but beware: *the shell performs the assignments from right to left.* Thus, the following command line results in the variable "A" acquiring the value "abc":

```
A=$B B=abc
```

The following are examples of simple assignments. Double quotation marks around the right-hand side allow spaces, tabs, semicolons, and newlines to be included in a string, while also allowing variable substitution (also known as "parameter substitution") to occur. This means that references to positional parameters and other variable names that are prefixed by a dollar sign ($) are replaced by the corresponding values, if any. Single quotation marks inhibit variable substitution:

```
MAIL=/usr/mail/gas
echovar="echo $1 $2 $3 $4"
stars=*****
asterisks='$stars'
```

In the above example, the variable *echovar* has as its value the string consisting of the values of the first four positional parameters, separated by spaces, plus the string "echo". No quotation marks are needed around the string of asterisks being assigned to *stars* because pattern matching (expansion of star, the question mark, and brackets) does not apply in this context. Note that the value of $*asterisks* is the literal string "$stars", *not* the string "*****", because the single quotation marks inhibit substitution.

In assignments, spaces are not re-interpreted after variable substitution, so that the following example results in *$first* and *$second* having the same value:

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

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

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

Here, the **echo** command prints:

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

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

```
test of variables.
```

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

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

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

MAIL — The pathname of a file where your mail is deposited. If MAIL is set, then the shell checks to see if anything has been added to the file it names and announces the arrival of new mail each time you return to command level (e.g., by leaving the editor). MAIL is not set automatically; if desired, it should be set (and optionally "exported") in the user's *.pwfile.* (The **export** command and *.profile* file are discussed later in this chapter.) (The presence of mail in the standard mail file is also announced at login, regardless of whether MAIL is set.)

MAILCHECK This parameter specifies how often (in seconds) the shell will check for the arrival of mail in the files specified by the MAILPATH or MAIL parameters. The default value is 600 seconds (10 minutes). If set to 0, the shell will check before each prompt.

MAILPATH A colon (:) separated list of filenames. If this parameter is set, the shell informs the user of the arrival of mail in any of the specified files. Each file name can be followed by % and a message that will be printed when the modification time changes. The default message is *you have mail.*

SHACCT If this parameter is set to the name of a file writable by the user, the shell will write an accounting record in the file for each shell procedure executed. Accountingroutines such as *acctcom*(C) and *accton*(C) can be used to anaylze the data collected.

SHELL When the shell is invoked, it scans the environment for this name. If it is found and there is an 'r' in the file name part of its value, the shell becomes a restricted shell.

PATH The variable that specifies the search path used by the shell in finding commands. Its value is an ordered list of directory pathnames separated by colons. The shell initializes PATH to the list *:/bin:/usr/bin* where a null argument appears in front of the first colon. A null anywhere in the path list represents the current directory. On some systems, a search of the current directory is *not* the default and the PATH variable is initialized instead to */bin:/usr/bin.* If you wish to search your current directory last, rather than first, use:

```
PATH=/bin:/usr/bin:
```

Below, the two colons together represent a colon followed by a null, followed by a colon, thus naming the current directory. You could possess a personal directory of commands (say, *$HOME/bin*) and cause it to be searched *before* the other three directories by using:

PATH=$HOME/bin::/bin:/usr/bin

"PATH" is normally set in your *.profile* file.

CDPATH — This variable defines the search path for the directory containing **arg**. Alternative directory names are separated by a colon (:). The default path is <null> (specifying the current directory). The current directory is specified by a null path name, which can appear immediately after the equal sign or between the colon delimiters anywhere else in the path list. If **arg** begins with a / then the search path is not used. Otherwise, each directory in the path is searched for **arg**.

PS1 — The variable that specifies what string is to be used as the primary *prompt* string. If the shell is interactive, it prompts with the value of PS1 when it expects input. The default value of PS1 is "$ " (a dollar sign ($) followed by a blank).

PS2 — The variable that specifies the secondary prompt string. If the shell expects more input when it encounters a newline in its input, it prompts with the value of PS2. The default value for this variable is "> " (a greater-than symbol followed by a space).

In general, you should be sure to **export** all of the above variables so that their values are passed to all shells created from your login. Use **export** at the end of your *.profile* file. An example of an **export** statement follows:

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

4.3.3 Predefined Special Variables

Several variables have special meanings; the following are set *only* by the shell:

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

```
sh cmd a b c
```

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

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

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

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

The following example illustrates the recommended practice of creating temporary files; note that the directories */usr* and */usr/tmp* are cleared out if the system is rebooted.

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

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

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

4.4 The Shell State

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

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

4.4.1 Changing Directories

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

For example, the first sequence below copies the file */etc/passwd* to */usr/you/passwd* ; the second example first changes directory to */etc* and then copies the file:

```
cp /etc/passwd /usr/you/passwd
(cd /etc; cp passwd /usr/you/passwd)
```

Note the use of parentheses. Both command lines have the same effect.

If the shell is reading its commands from a terminal, and the specified directory does not exist (or some component cannot be searched), spelling correction is applied to each component of *directory*, in a search for the "correct" name. The shell then asks whether or not to try and change directory to the corrected directory name; an answer of *n* means "no", and anything else is taken as "yes."

4.4.2 The .profile File

The file named *.profile* is read each time you login to XENIX. It is normally used to execute special one-time-only commands and to set and export variables to all later shells. Only after commands are read and executed from *.profile*, does the shell read commands from the standard input– usually the terminal.

4.4.3 Execution Flags

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

```
set -xv
```

The same flags may be turned *off* by entering:

```
set +xv
```

These two flags have the following meaning:

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

−x
: Commands and their arguments are printed as they are executed. (Shell control commands, such as **for, while**, etc., are not printed, however.) Note that **−x** causes a trace of only those commands that are actually executed, whereas **−v** prints each line of input until a syntax error is detected.

The **set** command is also used to set these and other flags within shell procedures.

4.5 A Command's Environment

All variables and their associated values that are known to a command at the beginning of its execution make up its *environment*. This environment includes variables that the command inherits from its parent process and variables specified as *keyword parameters* on the command line that invokes the command.

The variables that a shell passes to its child processes are those that have been named as arguments to the **export** command. The **export** command places the named variables in the environments of both the shell *and* all its future child processes.

Keyword parameters are variable-value pairs that appear in the form of assignments, normally *before* the procedure name on a command line. Such variables are placed in the environment of the procedure being invoked. Forexample:

```
#     keycommand
echo $a $b
```

This is a simple procedure that echoes the values of two variables. If it is invoked as:

```
a=key1 b=key2 keycommand
```

then theresultingoutput is:

```
key1 key2
```

Keyword parameters are *not* counted as arguments to the procedure and do not affect $#.

A procedure may access the value of any variable in its environment. However, if changes are made to the value of a variable, these changes are not reflected in the environment; they are local to the procedure in question. In order for these changes to be placed in the environment that the procedure passes to *its* child processes, the variable must be named as an argument to the **export** command within that procedure. To obtain a list of variables that have been made exportable from the current shell, enter:

```
export
```

You will also get a list of variables that have been made **readonly.** To get a list of name-value pairs in the current environment, enter either:

```
printenv
```

or

```
env
```

4.6 Invoking the Shell

The shell is a command and may be invoked in the same way as any other command:

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

sh -v *proc* [*arg*. . .]

This is equivalent to putting "set –v" at the beginning of *proc*. It can be used in the same way for the **–x**, **–e**, **–u**, and **–n** flags.

proc [*arg*. . .]

If *proc* is an executable file, and is not a compiled executable program, the effect is similar to that of:

```
sh proc args
```

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

4.7 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 arguments to the left; i.e., the value of $1 is thrown away, $2 replaces $1, $3 replaces $2, and so on. The highest-numbered positional parameter becomes *unset* ($0 is never shifted). For example, in the shell procedure *ripple* below, **echo** writes its arguments to the standard output.

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

Lines that begin with a number sign (#) are comments. The looping command, **while**, is discussed in "Conditional Looping: while and until" in this chapter. If the procedure were invoked with:

```
ripple a b c
```

it would print:

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

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

```
echo $*
```

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

```
wc $*
```

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

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

Sometimes, command lines are built inside a shell procedure. In this case, it is sometimes useful to have the shell rescan the command line after all the initial substitutions and expansions have been performed. The special command **eval** is available for this purpose. **eval** takes a command line as its argument and simply rescans the line, performing any variable or command substitutions that are specified. Consider the following (simplified) situation:

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

This segment of code results in the execution of the command line:

```
who | wc -l
```

Uses of **eval** can be nested so that a command line can be evaluated several times.

4.8 Controlling the Flow of Control

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

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

A *command* is a simple command or any of the shell control commands described below. A *pipeline* is a sequence of one or more commands separated by vertical bars (|). In a pipeline, the standard output of each command but the last is connected (by a *pipe*) to the standard input of the next command. Each command in a pipeline is run separately; the shell waits for the last command to finish. The exit status of a pipeline is the exit status of last process in the pipeline.

A *command list* is a sequence of one or more pipelines separated by a semicolon (;), an ampersand (&), an "and-if" symbol (&&), or an "or-if" (||) symbol, and optionally terminated by a semicolon or an ampersand. A semicolon causes sequential execution of the previous pipeline. This means that the shell waits for the pipeline to finish before reading the next pipeline. On the other hand, the ampersand (&) causes asynchronous background execution of the preceding pipeline. Thus, both sequential and background execution are allowed. A background pipeline continues execution until it terminates voluntarily, or until its processes are killed.

Other uses of the ampersand include off-line printing, background compilation, and generation of jobs to be sent to other computers. For example, if you enter:

```
nohup cc prog.c&
```

You may continue working while the C compiler runs in the background. A command line ending with an ampersand is immune to interrupts or quits that you might generate by typing **INTERRUPT** or **QUIT**. However, Ctrl-d *will* abort the command if you are operating over a dial-up line or have *stty hupcl*. In this case, it is wise to make the command immune to hang-ups (i.e., logouts) as well. The **nohup** command is used for this purpose. In the above example without **nohup**, if you log out from a dial-up line while **cc** is still executing, **cc** will be killed and your output will disappear.

The ampersand operator should be used with restraint, especially on heavily-loaded systems. Other users will not consider you a good citizen if you start up a large number of background processes without a compelling reason for doing so.

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

```
cmd1 || cmd2
```

the first command, *cmd1*, is executed and its exit status examined. Only if *cmd1* fails (i.e., has a nonzero exit status) is *cmd2* executed. Thus, this is a more terse notation for:

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

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

```
cmd1 && cmd2
```

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

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

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

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

Note that a space is needed after the left brace and that a semicolon should appear before the right brace.

4.8.1 Using the if Statement

The shell provides structured conditional capability with the **if** command. The simplest **if** command has the following form:

if *command-list*
then *command-list*
fi

The command list following the **if** is executed and if the last command in the list has a zero exit status, then the command list that follows **then** is executed. The word **fi** indicates the end of the **if** command.

To cause an alternative set of commands to be executed when there is a nonzero exit status, an **else** clause can be given with the following structure:

if *command-list*
then *command-list*
else *command-list*
fi

Multiple tests can be achieved in an **if** command by using the **elif** clause, although the **case** statement may be better for large numbers of tests. For example:

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

The above example is executed as follows: if the value of the first positional parameter is a filename (-f), then print that file; if not, then check to see if it is the name of a directory (-d). If so, change to that directory (cd) and print all the files there (pr *). Otherwise, **echo** the error message.

The **if** command may be nested (but be sure to end each one with a **fi**). The newlines in the above examples of **if** may be replaced by semicolons.

The exit status of the **if** command is the exit status of the last command executed in any **then** clause or **else** clause. If no such command was executed, **if** returns a zero exit status.

Note that an alternate notation for the **test** command uses brackets to enclose the expression being tested. For example, the previous example might have been written as follows:

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

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

4.8.2 Using the case Statement

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

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

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

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

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

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

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

4.8.3 Conditional Looping: while and until

A **while** command has the general form:

> **while** *command-list*
> **do**
>   *command-list*
> **done**

The commands in the first *command-list* are executed, and if the exit status of the last command in that list is zero, then the commands in the second *command-list* are executed. This sequence is repeated as long as the exit status of the first *command-list* is zero. A loop can be executed as long as the first command-list returns a nonzero exit status by replacing **while** with **until**.

Any newline in the above example may be replaced by a semicolon. The exit status of a **while** (or **until**) command is the exit status of the last command executed in the second *command-list*. If no such command is executed, **while** (or **until**) has a zero exit status.

4.8.4 Looping Over a List: for

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

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

Here *word-list* is a list of strings separated by blanks. The commands in the *command-list* are executed once for each word in the *word-list*. *Variable* takes on as its value each word from the word list, in turn. The word list is fixed after it is evaluated the first time. For example, the following **for** loop causes each of the C source files *xec.c*, *cmd.c*, and *word.c* in the current directory to be compared with a file of the same name in the directory */usr/src/cmd/sh*:

```
for CFILE in xec cmd word
do     diff $CFILE.c /usr/src/cmd/sh/$CFILE.c
done
```

Note that the first occurrence of CFILE immediately after the word **for** has no preceding dollar sign, since the name of the variable is wanted and not its value.

You can omit the "**in** *word-list*" part of a **for** command; this causes the current set of positional parameters to be used in place of word-list. This is useful when writing a command that performs the same set of commands for each of an unknown number of arguments.

As an example, create a file named *echo2* that contains the following shell script:

```
for word
do echo $word$word
done
```

Give *echo2* execute status:

```
chmod +x echo2
```

Now type the following command:

```
echo2 ma pa bo fi yo no so ta
```

The output from this command is:

```
mama
papa
bobo
fifi
yoyo
nono
soso
tata
```

4.8.5 Loop Control: break and continue

The **break** command can be used to terminate execution of a **while** or a **for** loop. The **continue** command immediately starts the execution of the next iteration of the loop. These commands are effective only when they appear between **do** and **done.**

The **break** command terminates execution of the smallest (i.e., innermost) enclosing loop, causing execution to resume after the nearest following unmatched **done.** Exit from *n* levels is obtained by **break** *n.*

The **continue** command causes execution to resume at the nearest enclosing **for, while**, or **until** statement, i.e., the one that begins the innermost loop containing the **continue.** You can also specify an argument *n* to **continue** and execution will resume at the *n*th enclosing loop:

```
# This procedure is interactive.
# "Break" and "continue" commands are used
# to allow the user to control data entry.
while true #loop forever
do     echo "Please enter data"
       read response
       case "$response" in
       "done")     break
              # no more data
              ;;
       "")    # just a carriage return,
              # keep on going
              continue
              ;;
       *)     # process the data here
              ;;
       esac
done
```

4.8.6 End-of-File and exit

When the shell reaches the end-of-file in a shell procedure, it terminates execution, returning to its parent the exit status of the last command executed prior to the end-of-file. The top level shell is terminated by typing a Ctrl-d (which logs the user out of XENIX).

The **exit** command simulates an end-of-file, setting the exit status to the value of its argument, if any. Thus, a procedure can be terminated normally by placing "exit 0" at the end of the file.

4.8.7 Command Grouping: Parentheses and Braces

There are two methods for grouping commands in the shell: parentheses and braces. Parentheses cause the shell to create a subshell that reads the enclosed commands. Both the right and left parentheses are recognized wherever they appear in a command line–they can appear as literal parentheses *only* when enclosed in quotation marks. For example, if you enter:

```
garble(stuff)
```

the shell prints an error message. Quoted lines, such as:

```
garble"("stuff")"
"garble(stuff)"
```

are interpreted correctly. Other quoting mechanisms are discussed in "Quoting Mechanisms" in this chapter.

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

The current environment is passed to the subshell and variables that are exported in the current shell are also exported in the subshell. Thus:

```
CURRENTDIR=`pwd`; cd /usr/docs/otherdir;
nohup nroff doc.n > doc.out&; cd $CURRENTDIR
```

and

```
(cd /usr/docs/otherdir; nohup nroff doc.n > doc.out&)
```

accomplish the same result: */usr/docs/otherdir/doc.n* is processed by *nroff* and the output is saved in */usr/docs/otherdir/doc.out*. (Note that **nroff** is a command available in the XENIX Text Processing System.) However, the second example automatically puts you back in your original working directory. In the second example above, blanks or newlines surrounding the parentheses are allowed but not necessary. When entering a command line at your terminal, the shell will prompt with the value of the shell variable PS2 if an end parenthesis is expected.

Braces ({ and }) may also be used to group commands together. Both the left and the right brace are recognized *only* if they appear as the first (unquoted) word of a command. The opening brace may be followed by a newline (in which case the shell prompts for more input). Unlike parentheses, no subshell is created for braces: the enclosed commands are simply read by the shell. The braces are convenient when you wish to use the (sequential) output of several commands as input to one command.

The exit status of a set of commands grouped by either parentheses or braces is the exit status of the last enclosed executed command.

4.8.8 Defining Functions

The shell includes a function definition capability. Functions are like shell scripts or procedures except that they reside in memory and so are executed by the shell process, not by a separate process. The basic form is:

name ( ) {*list*;}

list can include any of the commands previously discussed. Functions can be defined in one section of a shell script to be called as many times as needed, making them easier to write and maintain. Here is an example of a function called "getyn":

```
# Prompt for yes or no answer - returns non-zero for no
getyn( )     {
        while echo "0* (y/n)? c">&2
        do    read yn rest
              case $yn in
              [yY]) return 0                        ;;
              [nN]) return 1                        ;;
              *)    echo "Please answer y or n" >&2 ;;
              esac
        done
}
```

In this example, the function appends a "(y/n)?" to the output and accepts "Y", "y", "n" or "N" as input, returning a 0 or 1. If the input is anything else, the function prompts the user for the correct input. (Echo should never fail, so the while-loop is effectively infinite.)

Functions are used just like other commands; an invocation of *getyn* might be:

```
getyn "Do you wish to continue" || exit
```

However, unlike other commands, the shell positional parameters **$1, $2,** ..., are set to the arguments of the function. Since an exit in a function will terminate the shell procedure, the return command should be used to return a value back to the procedure.

4.8.9 Input/Output Redirection and Control Commands

The shell normally does *not* fork and create a new shell when it recognizes the control commands (other than parentheses) described above. However, each command in a pipeline is run as a separate process in order to direct input to or output from each command. Also, when redirection of input or output is specified explicitly to a control command, a separate process is spawned to execute that command. Thus, when **if, while, until, case,** and **for** are used in a pipeline consisting of more than one command, the shell forks and a subshell runs the control command. This has two implications:

1. Any changes made to variables within the control command are not effective once that control command finishes (this is similar to the effect of using parentheses to group commands).

2. Control commands run slightly slower when redirected, because of the additional overhead of creating a shell for the control command.

4.8.10 Transfer Between Files: The Dot (.) Command

A command line of the form:

```
. proc
```

causes the shell to read commands from *proc* without spawning a new process. Changes made to variables in *proc* are in effect after the dot command finishes. This is a good way to gather a number of shell variable initializations into one file. A common use of this command is to reinitialize the top level shell by reading the *.profile* file with:

```
. .profile
```

4.8.11 Interrupt Handling: trap

Shell procedures can use the **trap** command to disable a signal (cause it to be ignored), or redefine its action. The form of the **trap** command is:

trap *arg signal-list*

Here *arg* is a string to be interpreted as a command list and *signal-list* consists of one or more signal numbers as described in **signal** (S) in the XENIX *Reference Manual*. The most important of these signals follow:

Number	Signal
0	Exit from the shell
1	HANGUP
2	INTERRUPT character (DELETE or RUB OUT)
3	QUIT (Ctrl-\)
9	KILL (cannot be caught or ignored)
11	Segmentation violation (cannot be caught or ignored)
15	Software termination signal

The commands in *arg* are scanned at least once, when the shell first encounters the **trap** command. Because of this, it is usually wise to use single rather than double quotation marks to surround these commands. The former inhibit immediate command and variable substitution. This becomes important, for instance, when one wishes to remove temporary files and the names of those files have not yet been determined when the trap command is first read by the shell. The following procedure will print the name of the current directory in the user information as to how much of the job was done:

```
trap 'echo Directory was `pwd` when interrupted' 2 3 15
for i in /bin /usr/bin /usr/gas/bin
do
        cd $i
            # commands to be executed in directory $i here
done
```

Beware that the same procedure with double rather than single quotation marks does something different. The following prints the name of the directory from which the procedure was first executed:

```
trap "echo Directorywas `pwd` wheninterrupted" 2 3 15
```

A signal 11 can never be trapped, because the shell itself needs to catch it to deal with memory allocation. Zero is interpreted by the **trap** command as a signal generated by exiting from a shell. This occurs either with an **exit** command, or by "falling through" to the end of a procedure. If *arg* is not specified, then the action taken upon receipt of any of the signals in the signal list is reset to the default system action. If *arg* is an explicit null string (˝ or ""), then the signals in the signal list are ignored by the shell.

The **trap** command is most frequently used to make sure that temporary files are removed upon termination of a procedure. The preceding example would be written more typically as follows:

```
temp=$HOME/temp/$$
trap 'rm -F $temp; exit' 0 1 2 3 15
ls > $temp
  # commands that use $temp here
```

In this example, whenever signal 1 (hangup), 2 (interrupt), 3 (quit), or 15(terminate) is received by the shell procedure, or whenever the shell procedure is about to exit, the commands enclosed between the single quotation marks are executed. The **exit** command must be included, or else the shell continues reading commands where it left off when the signal was received.

Sometimes the shell continues reading commands after executing trap commands. The following procedure takes each directory in the current directory, changes to that directory, prompts with its name, and executes commands typed at the terminal until an end-of-file (Ctrl-D) or an interrupt is received. An end-of-file causes the **read** command to return a onzero exit status, and thus the **while** loop terminates and the next directory cycle is initiated. An interrupt is ignored while executing the requested commands, but causes termination of the procedure when it is waiting for input:

```
d=`pwd`
for i in *
do    if test -d $d/$i
      then cd $d/$i
            while echo "$i:"
                  trap exit 2
                  read x
            do    trap : 2
                  # ignore interrupts
                  eval $x
            done
      fi
done
```

Several **traps** may be in effect at the same time: if multiple signals are received simultaneously, they are serviced in numerically ascending order. To determine which traps are currently set, enter:

```
trap
```

It is important to understand some things about the way in which the shell implements the **trap** command. When a signal (other than 11) is received by the shell, it is passed on to whatever child processes are currently executing. When these (synchronous) processes terminate, normally or abnormally, the shell polls any traps that happen to be set and executes the appropriate **trap** commands. This process is straightforward, except in the case of traps set at the command (outermost, or login) level. In this case, it is possible that no child process is running, so before the shell polls the traps, it waits for the termination of the first process spawned *after* the signal was received.

When a signal is redefined in a shell script, this does not redefine the signal for programs invoked by that script; the signal is merely passed along. A disabled signal is not passed.

For internal commands, the shell normally polls traps on completion of the command. An exception to this rule is made for the **read** command, for which traps are serviced immediately, so that **read** can be interrupted while waiting for input.

4.9 Special Shell Commands

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

Several of the special commands have already been described because they affect the flow of control. They are dot (**.**), **break**, **continue**, **exit**, and **trap.** The **set** command is also a special command. Descriptions of the remaining special commands are given here:

: The null command. This command does nothing and can be used to insert comments in shell procedures. Its exit status is zero (true). Its utility as a comment character has largely been supplanted by the number sign (#) which can be used to insert comments to the end-of-line. Beware: any arguments to the null command are parsed for syntactic correctness; when in doubt, quote such arguments. Parameter substitution takes place, just as in other commands.

cd *arg* — Make *arg* the current directory. If *arg* is not a valid directory, or the user is not authorized to access it, a nonzero exit status is returned. Specifying **cd** with no *arg* is equivalent to entering "cd$HOME" which takes you to your home directory.

exec *arg* ... — If *arg* is a command, then the shell executes the command without forking and returning to the current shell. This is effectively a "goto" and no new process is created. Input and output redirection arguments are allowed on the command line. If *only* input and output redirection arguments appear, then the input and output of the shell itself are modified accordingly.

hash [*−r*] *name* — For each *name*, the location in the search path of the command specified by *name* is determined and remembered by the shell. The **−r** option causes the shell to forget all remembered locations. If no arguments are given, information about remembered commands is presented. *Hits* is the number of times a command has been invoked by the shell process. *Cost* is a measure of the work required to locate a command in the search path. There are certain situations which require that the stored location of a command be recalculated. Commands for which this will be done are indicated by an asterisk (*) adjacent to the *hits* information. *Cost* will be incremented when the recalculation is done.

newgrp *arg* ... — The **newgrp** command is executed, replacing the shell. **Newgrp** in turn creates a new shell. Beware: only environment variables will be known in the shell created by the **newgrp** command. Any variables that were exported will no longer be marked as such.

pwd — Print the current working directory. See *pwd*(C) for usage and description.

read *var* ... — One line (up to a newline) is read from the standard input and the first word is assigned to the first variable, the second word to the second variable, and so on. All words left over are assigned to the *last* variable. The exit status of **read** is zero unless an end-of-file is read.

readonly *var* ...

The specified variables are made **readonly** so that no subsequent assignments may be made to them. If no arguments are given, a list of all **readonly** and of all exported variables is given.

return *n*

Causes a function to exit with the return value specified by *n*. If *n* is omitted, the return status is that of the last command executed.

times

The accumulated user and system times for processes run from the current shell are printed.

type *name*

For each *name*, indicate how it would be interpreted if used as a command name.

ulimit [-f] *n*

This imposes a size limit of *n* blocks on files written. The **-f** flag imposes a size limit of *n* blocks on files written by child processes (files of any size may be read). With no argument, the current limit is printed. If no option is given and a number is specified, **-f** is assumed.

umask *nnn*

The user file creation mask is set to *nnn*. If *nnn* is omitted, then the current value of the mask is printed. This bit-mask is used to set the default permissions when creating files. For example, an octal umask of 137 corresponds to the following bit-mask and permission settings for a newly created file:

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

See *umask*(C) in the XENIX *Reference Manual* for information on the value of *nnn*.

unset *name* — For each *name*, remove the corresponding variable or function. The variables **PATH**, **PS1**, **PS2**, **MAILCHECK** and **IFS** cannot be unset.

wait *n* — The shell waits for all currently active child processes to terminate. If *n* is specified, the shell waits for the specified process to terminate. The exit status of *wait* is always zero if *n* is not given; otherwise it is the exit status of child *n*.

4.10 Creation and Organization of Shell Procedures

A shell procedure can be created in two simple steps. The first is building an ordinary text file. The second is changing the *mode* of the file to make it *executable*, thus permitting it to be invoked by:

```
proc args
```

rather than

```
sh proc args
```

The second step may be omitted for a procedure to be used once or twice and then discarded, but is recommended for frequently-used ones. For example, create a file named *mailall* with the following contents:

```
LETTER=$1
shift
for i in $*
do mail $i < $LETTER
done
```

Next enter:

```
chmod +x mailall
```

The new command might then be invoked from within the current directory by entering:

```
mailall letter joe bob
```

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

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

Shell procedures are often used by users running the **csh**. However, if the first character of the procedure is a # (comment character), the **csh** assumes the procedure is a **csh** script, and invokes */bin/csh* to execute it. Always start sh procedures with some other character if **csh** users are to run the procedure at any time. This invokes the standard shell */bin/sh*.

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

Many users prefer writing shell procedures to writing programs in C or other traditional languages. This is true for several reasons:

1. A shell procedure is easy to create and maintain because it is only a file of ordinary text.

2. A shell procedure has no corresponding object program that must be generated and maintained.

3. A shell procedure is easy to create quickly, use a few times, and then remove.

4. Because shell procedures are usually short in length, written in a high-level programming language, and kept only in their source-language form, they are generally easy to find, understand, and modify.

By convention, directories that contain only commands and shell procedures are named *bin*. This name is derived from the word "binary", and is used because compiled and executable programs are often called "binaries" to distinguish them from program source files. Most groups of users sharing common interests have one or more *bin* directories set up to hold common procedures. Some users have their PATH variable list several such directories. Although you can have a number of such directories, it is unwise to go overboard: it may become difficult to keep track of your environment and efficiency may suffer.

4.11 More About Execution Flags

There are several execution flags available in the shell that can be useful in shell procedures:

–e This flag causes the shell to exit immediately if any command that it executes exits with a nonzero exit status. This flag is useful for shell procedures composed of simple command lines; it is not intended for use in conjunction with other conditional constructs.

–u This flag causes unset variables to be considered errors when substituting variable values. This flag can be used to effect a global check on variables, rather than using conditional substitution to check each variable.

–t This flag causes the shell to exit after reading and executing the commands on the remainder of the current input line. This flag is typically used by C programs which call the shell to execute a single command.

–n This is a "don't execute" flag. On occasion, one may want to check a procedure for syntax errors, but not execute the commands in the procedure. Using "set–nv" at the beginning of a file will accomplish this.

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

4.12 Supporting Commands and Features

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

4.12.1 Conditional Evaluation: test

The **test** command evaluates the expression specified by its arguments and, if the expression is true, **test** returns a zero exit status. Otherwise, a nonzero (false) exit status is returned. **test** also returns a nonzero exit status if it has no arguments. Often it is convenient to use the **test** command as the first command in the command list following an **if** or a **while**. Shell variables used in **test** expressions should be enclosed in double quotation marks if there is any chance of their being null or not set.

The square brackets may be used as an alias to **test**, so that:

[*expression*]

has the same effect as:

test *expression*

Note that the spaces before and after the *expression* in brackets are essential.

The following is a partial list of the options that can be used to construct a conditional expression:

−r *file*	True if the named file exists and is readable by the user.
−w *file*	True if the named file exists and is writable by the user.
−x *file*	True if the named file exists and is executable by the user.
−s *file*	True if the named file exists and has a size greater than zero.
−d *file*	True if the named file is a directory.
−f *file*	True if the named file is an ordinary file.
−z *s1*	True if the length of string *s1* is zero.
−n *s1*	True if the length of the string *s1* is nonzero.
−t *fildes*	True if the open file whose file descriptor number is *fildes* is associated with a terminal device. If *fildes* is not specified, file descriptor 1 is used by default.
s1 = *s2*	True if strings *s1* and *s2* are identical.
s1 != *s2*	True if strings *s1* and *s2* are *not* identical.
s1	True if *s1* is *not* the null string.
n1 **−eq** *n2*	True if the integers *n1* and *n2* are algebraically equal; other algebraic comparisons are indicated by **−ne** (not equal), **−gt** (greater than), **−ge** (greater than or equal to), **−lt** (less than), and **−le** (less than or equal to).

These may be combined with the following operators:

!	Unary negation operator.
–a	Binary logical AND operator.
–o	Binary logical OR operator; it has lower precedence than the logical AND operator (–a).
(*expr*)	Parentheses for grouping; they must be escaped to remove their significance to the shell. In the absence of parentheses, evaluation proceeds from left to right.

Note that all options, operators, filenames, etc. are separate arguments to **test.**

4.12.2 Echoing Arguments

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

echo [*options*] [*args*]

echo copies its arguments to the standard output, each followed by a single space, except for the last argument, which is normally followed by a newline. You can use it to prompt the user for input, to issue diagnostics in shell procedures, or to add a few lines to an output stream in the middle of a pipeline. Another use is to verify the argument list generation process before issuing a command that does something drastic.

You can replace the ls command with

```
echo *
```

because the latter is faster and prints fewer lines of output.

The **–n** option to **echo** removes the newline from the end of the echoed line. Thus, the following two commands prompt for input and then allow entering on the same line as the prompt:

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

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

4.12.3 Expression Evaluation: expr

The **expr** command provides arithmetic and logical operations on integers and some pattern-matching facilities on its arguments. It evaluates a single expression and writes the result on the standard output; **expr** can be used inside grave accents to set a variable. Some typical examples follow:

```
#     increment $A
A=`expr $a + 1`
#     put third through last characters of
#     $1 into substring
substring=`expr "$1" : '..\(.*\)' `
#     obtain length of $1
c=`expr "$1" : '.*' `
```

The most common uses of **expr** are in counting iterations of a loop and in using its pattern-matching capability to pick apart strings.

4.12.4 True and False

The **true** and **false** commands perform the functions of exiting with zero and nonzero exit status, respectively. The **true** and **false** commands are often used to implement unconditional loops. For example, you might enter:

```
while true
do echo forever
done
```

This will echo "forever" on the screen until an INTERRUPT is entered.

4.12.5 In-Line Input Documents

Upon seeing a command line of the form:

command << *eofstring*

where *eofstring* is any arbitrary string, the shell will take the subsequent lines as the standard input of *command* until a line is read consisting only of *eofstring*. (By appending a minus (–) to the input redirection symbol (<<), leading spaces and tabs are deleted from each line of the input document before the shell passes the line to *command*.)

The shell creates a temporary file containing the input document and performs variable and command substitution on its contents before passing it to the command. Pattern matching on filenames is performed on the arguments of command lines in command substitutions. In order to prohibit all substitutions, you may quote any character of *eofstring*:

```
command <<\eofstring
```

The in-line input document feature is especially useful for small amounts of input data, where it is more convenient to place the data in the shell procedure than to keep it in a separate file. For instance, you could enter:

```
cat <<- xx
        This message will be printed on the
        terminal with leading tabs and spaces
        removed.
xx
```

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

4.12.6 Input / Output Redirection Using File Descriptors

We mentioned above that a command occasionally directs output to some file associated with a file descriptor other than 1 or 2. In languages such as C, one can associate output with any file descriptor by using the **write** (S) system call (see the XENIX *Reference Manual*). The shell provides its own mechanism for creating an output file associated with a particular file descriptor. By entering:

fd1 >& *fd2*

where *fd1* and *fd2* are valid file descriptors, one can direct output that would normally be associated with file descriptor *fd1* to the file associated with *fd2*. The default value for *fd1* and *fd2* is 1. If, at run time, no file is associated with *fd2*, then the redirection is void. The most common use of this mechanism is that of directing standard error output to the same file as standard output. This is accomplished by entering:

```
command 2>&1
```

If you wanted to redirect both standard output and standard error output to the same file, you would enter:

```
command 1> file 2>&1
```

The order here is significant: first, file descriptor 1 is associated with *file*; then file descriptor 2 is associated with the same file as is currently associated with file descriptor 1. If the order of the redirections were reversed, standard error output would go to the terminal, and standard output would go to *file*, because at the time of the error output redirection, file descriptor 1 still would have been associated with the terminal.

This mechanism can also be generalized to the redirection of standard input. You could enter:

fda <& *fdb*

to cause both file descriptors *fda* and *fdb* to be associated with the same input file. If *fda* or *fdb* is not specified, file descriptor 0 is assumed. Such input redirection is useful for a command that uses two or more input sources.

4.12.7 Conditional Substitution

Normally, the shell replaces occurrences of $*variable* by the string value assigned to *variable*, if any. However, there exists a special notation to allow conditional substitution, dependent upon whether the variable is set or not null. By definition, a variable is set if it has ever been assigned a value. The value of a variable can be the null string, which may be assigned to a variable in 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 upon whether a variable is set and not null. Note that the meaning of braces in these expressions differs from their meaning when used in grouping shell commands. *Parameter* as used below refers to either a digit or a variable name.

${*variable*:-*string*} — If *variable* is set and is nonnull, then substitute the value $*variable* in place of this expression. Otherwise, replace the expression with *string*. Note that the value of *variable* is *not* changed by the evaluation of this expression.

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

${*variable* :?*string*}

If *variable* is set and is nonnull, then substitute the value of *variable* for the expression. Otherwise, print a message of the form

variable: *string*

and exit from the current shell. (If the shell is the login shell, it is not exited.) If *string* is omitted in this form, then the message

variable: parameter null or not set

is printed instead.

${*variable* :+*string*}

If *variable* is set and is nonnull, then substitute *string* for this expression. Otherwise, substitute the null string. Note that the value of *variable* is not altered by the evaluation of this expression.

These expressions may also be used without the colon. In this variation, the shell does not check whether the variable is null or not; it only checks whether the variable has ever been set.

The two examples below illustrate the use of this facility:

1. This example performs an explicit assignment to the PATH variable:

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

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

2. This example automatically assigns the HOME variable a value:

   ```
   cd ${HOME:='/usr/gas'}
   ```

   If HOME is set, and is not null, then change directory to it. Otherwise set HOME to the given value and change directory to it.

4.12.8 Invocation Flags

There are five flags that may be specified on the command line when invoking the shell. These flags may not be turned on with the **set** command:

-i If this flag is specified, or if the shell's input and output are both attached to a terminal, the shell is *interactive*. In such a shell, INTERRUPT (signal 2) is caught and ignored, and TERMINATE (signal 15) and QUIT (signal 3) are ignored.

-s If this flag is specified or if no input/output redirection arguments are given, the shell reads commands from standard input. Shell output is written to file descriptor 2. All remaining arguments specify the positional parameters.

-c When this flag is turned on, the shell reads commands from the first string following the flag. Remaining arguments are ignored.

-t When this flag is on, a single command is read and executed, then the shell exits. This flag is not useful interactively, but is intended for use with C programs.

-r If this flag is present the shell is a restricted shell (see rsh (C)).

4.13 Effective and Efficient Shell Programming

This section outlines strategies for writing efficient shell procedures, ones that do not waste resources in accomplishing their purposes. The primary reason for choosing a shell procedure to perform a specific function is to achieve a desired result at a minimum human cost. Emphasis should always be placed on simplicity, clarity, and readability, but efficiency can also be gained through awareness of a few design strategies. In many cases, an effective redesign of an existing procedure improves its efficiency by reducing its size, and often increases its comprehensibility. In any case, you should not worry about optimizing shell procedures unless they are intolerably slow or are known to consume an inordinate amount of a system's resources.

The same kind of iteration cycle should be applied to shell procedures as to other programs: write code, measure it, and optimize only the *few* important parts. The user should become familiar with the **time** command, which can be used to measure both entire procedures and parts thereof. Its use is strongly recommended; human intuition is notoriously unreliable when used to estimate timings of programs, even when the style of programming is a familiar one. Each timing test should be run several times, because the results are easily disturbed by variations in system load.

4.13.1 Number of Processes Generated

When large numbers of short commands are executed, the actual execution time of the commands may well be dominated by the overhead of creating processes. The procedures that incur significant amounts of such overhead are those that perform much looping, and those that generate command sequences to be interpreted by another shell.

If you are worried about efficiency, it is important to know which commands are currently built into the shell, and which are not. Here is the alphabetical list of those that are built in:

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

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

The user should always have at least a vague idea of the number of processes generated by a shell procedure. In the bulk of observed procedures, the number of processes created (not necessarily simultaneously) can be described by:

processes = (k*n) + c

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

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

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

```
:
#       split
trap 'rm temp$$; trap 0; exit' 0 1 2 3 15
start1=0 start2=0
b='[A-Za-z]'
cat > temp$$
            # read stdin into temp file
            # save original lengths of $1, $2
if test -s "$1"
then start1=`wc -l < $1`
fi
if test -s "$2"
then start2=`wc -l < $2`
fi
grep "$b" temp$$ >> $1
            # lines with letters onto $1
grep -v "$b" temp$$ | grep '[0-9]' >> $2
            # lines without letters onto $2
total=" `wc -l < temp$$` "
end1=" `wc -l < $1` "
end2=" `wc -l < $2` "
lost=" `expr $total - \($end1 - $start1\) \
- \($end2 - $start2\)` "
echo "$total read, $lost thrown away"
```

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

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

4.13.2 Number of Data Bytes Accessed

It is worthwhile to consider any action that reduces the number of bytes read or written. This may be important for those procedures whose time is spent passing data around among a few processes, rather than in creating large numbers of short processes. Some filters shrink their output, others usually increase it. It always pays to put the *shrinkers* first when the order is irrelevant. For instance, the second of the following examples is likely to be faster because the input to **sort** will be much smaller:

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

4.13.3 Shortening Directory Searches

Directory searching can consume a great deal of time, especially in those applications that utilize deep directory structures and long pathnames. Judicious use of **cd**, the *change directory* command, can help shorten long pathnames and thus reduce the number of directory searches needed. As an exercise, try the following commands:

```
ls -l /usr/bin/* >/dev/null
cd /usr/bin; ls -l * >/dev/null
```

The second command will run faster because of the fewer directory searches.

4.13.4 Directory-Search Order and the PATH Variable

The PATH variable is a convenient mechanism for allowing organization and sharing of procedures. However, it must be used in a sensible fashion, or the result may be a great increase in system overhead.

The process of finding a command involves reading every directory included in every pathname that precedes the needed pathname in the current PATH variable. As an example, consider the effect of invoking **nroff** (i.e., */usr/bin/nroff*) when the value of PATH is ":/bin:/usr/bin". The sequence of directories read is:

```
.
/
/bin
/
/usr
/usr/bin
```

This is a total of six directories. A long path list assigned to PATH can increase this number significantly.

The vast majority of command executions are of commands found in */bin* and, to a somewhat lesser extent, in */usr/bin*. Careless PATH setup may lead to a great deal of unnecessary searching. The following four examples are ordered from worst to best with respect to the efficiency of command searches:

```
:/usr/john/bin:/usr/localbin:/bin:/usr/bin
:/bin:/usr/john/bin:/usr/localbin:/usr/bin
:/bin:/usr/bin:/usr/john/bin:/usr/localbin
/bin::/usr/bin:/usr/john/bin:/usr/localbin
```

The first one above should be avoided. The others are acceptable and the choice among them is dictated by the rate of change in the set of commands kept in */bin* and */usr/bin*.

A procedure that is expensive because it invokes many short-lived commands may often be speeded up by setting the PATH variable inside the procedure so that the fewest possible directories are searched in an optimum order.

4.13.5 Good Ways to Set Up Directories

It is wise to avoid directories that are larger than necessary. You should be aware of several special sizes. A directory that contains entries for up to 30 files (plus the required . and ..) fits in a single disk block and can be searched very efficiently. One that has up to 286 entries is still a small directory; anything larger is usually a disaster when used as a working directory. It is especially important to keep login directories small, preferably one block at most. Note that, as a rule, directories never shrink. This is very important to understand, because if your directory ever exceeds either the 30 or 286 thresholds, searches will be inefficient; furthermore, even if you delete files so that the number of files is less than either threshold, the system will still continue to treat the directory inefficiently.

4.14 Shell Procedure Examples

The power of the XENIX shell command language is most readily seen by examining how many labor-saving XENIX utilities can be combined to perform powerful and useful commands with very little programming effort. This section gives examples of procedures that do just that. By studying these examples, you will gain insight into the techniques and shortcuts that can be used in programming shell procedures (also called "scripts"). Note the use of the null command (:) to begin each shell procedure and the use of the number sign (#) to introduce comments.

It is intended that the following steps be carried out for each procedure:

1. Place the procedure in a file with the indicated name.

2. Give the file execute permission with the **chmod** command.

3. Move the file to a directory in which commands are kept, such as your own *bin* directory.

4. Make sure that the path of the *bin* directory is specified in the PATH variable found in *.profile*.

5. Execute the named command.

BINUNIQ

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

This procedure determines which files are in both */bin* and */usr/bin*. It is done because files in */bin* will "override" those in */usr/bin* during most searches and duplicates need to be weeded out. If the */usr/bin* file is obsolete, then space is being wasted; if the */bin* file is outdated by a corresponding entry in */usr/bin* then the wrong version is being run and, again, space is being wasted. This is also a good demonstration of "sort | uniq" to find matches and duplications.

COPYPAIRS

```
:
#       Usage: copypairs file1 file2 ...
#       Copies file1 to file2, file3 to file4, ...
while test "$2" != ""
do
        cp $1 $2
        shift; shift
done
if test "$1" != ""
        then echo "$0: odd number of arguments" >&2
fi
```

This procedure illustrates the use of a **while** loop to process a list of positional parameters that are somehow related to one another. Here a **while** loop is much better than a **for** loop, because you can adjust the positional parameters with the **shift** command to handle related arguments.

COPYTO

```
:
#	Usage: copyto dir file ...
#	Copies argument files to "dir",
#	making sure that at least
#	two arguments exist, that "dir" is a directory,
#	and that each additional argument
#	is a readable file.
if test $# -lt 2
	then echo "$0: usage: copyto directory file ...">&2
elif test ! -d $1
	then echo "$0: $1 is not a directory";>&2
else	dir=$1; shift
	for eachfile
	do	cp $eachfile $dir
	done
fi
```

This procedure uses an **if** command with several parts to screen out improper usage. The **for** loop at the end of the procedure loops over all of the arguments to **copyto** but the first; the original **$1** is shifted off.

DISTINCT1

```
:
#	Usage: distinct1
#	Reads standard input and reports list of
#	alphanumeric strings that differ only in case,
#	giving lowercase form of each.
tr -cs 'A-Za-z0-9' '\012' | sort-u |\
tr 'A-Z' 'a-z' | sort | uniq -d
```

This procedure is an example of the kind of process that is created by the left-to-right construction of a long pipeline. Note the use of the backslash at the end of the first line as the line continuation character. It may not be immediately obvious how this command works. You may wish to consult **tr** (C), **sort** (C), and **uniq** (C) in the XENIX *Reference Manual* if you are completely unfamiliar with these commands. The **tr** command translates all characters except letters and digits into newline characters, and then squeezes out repeated newline characters. This leaves each string (in this case, any contiguous sequence of letters and digits) on a separate line. The **sort** command sorts the lines and emits only one line from any sequence of one or more repeated lines. The next **tr** converts everything to lowercase, so that identifiers differing only in case become identical. The output is sorted again to bring such duplicates together. The "uniq -d" prints (once) only those lines that occur more than once, yielding the desired list.

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

```
cmd1 | cmd2 | cmd3

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

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

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

DRAFT

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

Users often write this kind of procedure for convenience in dealing with commands that require the use of distinct flags that cannot be given default values that are reasonable for all (or even most) users.

EDFIND

```
:
#       Usage: edfind filearg
#       Finds the last occurrence in "file" of a line
#       whose beginning matches "arg", then prints
#       3 lines (the one before, the line itself,
#       and the one after)
ed - $1<< -EOF
        ?^$2?
        -,+p
        q
EOF
```

This illustrates the practice of using **ed** in-line input scripts into which the shell can substitute the values of variables.

EDLAST

```
:
#       Usage: edlast file
#       Prints the last line of file,
#       then deletes that line.
ed - $1 <<-\!
        $p
        $d
        w
        q
!
echo done
```

This procedure illustrates taking input from within the file itself up to the exclamation point (!). Variable substitution is prohibited within the input text because of the backslash.

FSPLIT

```
:
#       Usage: fsplit file1 file2
#       Reads standard input and divides it into 3 parts
#       by appending any line containing at least one letter
#       to file1, appending any line containing digits but
#       no letters to file2, and by throwing the rest away.
count=0 gone=0
while read next
do
        count="`expr $count + 1`"
        case "$next" in
        *[A-Za-z]*)
                echo "$next" >> $1 ;;
        *[0-9]*)
                echo "$next" >> $2 ;;
        *)
                gone="`expr $gone + 1`"
        esac
done
echo "$count lines read, $gone thrown away"
```

Each iteration of the loop reads a line from the input and analyzes it. The loop terminates only when **read** encounters an end-of-file. Note the use of the **expr** command.

Do not use the shell to read a line at a time unless you must because it can be an extremely slow process.

LISTFIELDS

```
:
grep $* | tr ":" "\012"
```

This procedure lists lines containing any desired entry that is given to it as an argument. It places any field that begins with a colon on a newline. Thus, if given the following input:

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

listfields will produce this:

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

Note the use of the **tr** command to transpose colons to linefeeds.

MKFILES

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

The *mkfiles* procedure uses output redirection to create zero-length files. The **expr** command is used for counting iterations of the **while** loop.

NULL

```
:
#       Usage: null files
#       Create each of the named files as an empty file.
for eachfile
do
        >$eachfile
done
```

This procedure uses the fact that output redirection creates the (empty) output file if a file does not already exist.

PHONE

```
:
#       Usage: phone initials ...
#       Prints the phone numbers of the
#       people with the given initials.
echo '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 database.

TEXTFILE

```
:
if test "$1" = "-s"
then
#       Return condition code
        shift
        if test -z "`$0 $*`" # check return value
        then
                exit 1
        else
                exit 0
        fi
fi

if test $# -lt 1
then    echo "$0: Usage: $0 [ -s ] file ..." 1>&2
        exit 0
fi

file $* | fgrep ' text' | sed 's/:  .*//'
```

To determine which files in a directory contain only textual information, *textfile* filters argument lists to other commands. For example, the following command line will print all the text files in the current directory:

```
pr `textfile *` | lpr
```

This procedure also uses an **−s** flag which silently tests whether any of the files in the argument list is a text file.

WRITEMAIL

```
:
#       Usage: writemail message user
#       If user is logged in,
#       writes message to terminal;
#       otherwise, mails it to user.
echo "$1" | { write "$2" || mail "$2" ;}
```

This procedure illustrates the use of command grouping. The message specified by $1 is piped to both the **write** command and, if **write** fails, to the **mail** command.

4.15 Shell Grammar

item: *word*
input-output
name = *value*

simple-command: *item*
simple-command item

command: *simple-command*
(*command-list*)
{ *command-list* }
for *name* **do** *command-list* **done**
for *name* **in** *word* **do** *command-list* **done**
while *command-list* **do** *command-list* **done**
until *command-list* **do** *command-list* **done**
case *word* **in** *case-part* **esac**
if *command-list* **then** *command-list else-part* **fi**

pipeline: *command*
pipeline | *command*

andor: *pipeline*
andor **&&** *pipeline*
andor **||** *pipeline*

command-list: *andor*
command-list **;**
command-list **&**
command-list **;** *andor*
command-list **&** *andor*

input-output: > *file*
< *file*
<< *word*
>> *file*
digit > *file*
digit < *file*
digit >> *file*

file: *word*
& *digit*
& –

case-part: *pattern* **)** *command-list* **;;**

pattern: *word*
pattern | *word*

else-part: **elif** *command-list* **then** *command-list else-part*
else *command-list*
empty

empty:

word: *a sequence of nonblank characters*

name: *a sequence of letters, digits, or underscores starting with a letter*

digit: **0 1 2 3 4 5 6 7 8 9**

Metacharacters and Reserved Words

1. Syntactic

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

2. Patterns

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

3. Substitution

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

4. Quoting

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

5. Reserved words

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

Replace this Page
with Tab Marked:

bc Calculator

Chapter 5

bc: A Calculator

5.1 Introduction

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

Common uses for **bc** are:

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

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

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

5.2 Demonstration

This demonstration is designed to show you:

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

A normal session with **bc** begins by invoking the program with the command:

```
bc
```

To exit **bc** enter:

quit

or press **Ctrl−d**. Once you have entered **bc**, you can use it very much like a normal calculator. As with the XENIX shell, commands are read as command-lines, so each line that you enter must be terminated by a **RETURN**. Throughout this chapter, the **RETURN** is implied at the end of each command line. Within **bc**, normal processing of other keys, such as **BACKSPACE** and **INTERRUPT**, also works.

For example, enter the simple integer 5:

5

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

5

Here *5* is a simple numeric expression. However, if you enter the expression:

5*5.25

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

26.25

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

(10*5)+50−(50/2)

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

The simple addition:

10.45+5.5555555

produces the output:

    16.0055555

Note how precision is retained in the above result.

The two-part multiplication:

    (8*9)*7

produces the answer:

    504

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

    a=100 ; b=5

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

    a*b; a+b

to produce:

    500
    105

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

    quit

or press **Ctrl- d**.

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

5.3 Tasks

This section describes how to perform common **bc** tasks. Mastery of these tasks should turn you into a competent **bc** user.

5.3.1 Computing with Integers

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

```
142857 + 285714
```

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

```
428571
```

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

```
+ - * / % ^
```

They indicate addition, subtraction, multiplication, division, modulo (remaindering), and exponentiation, respectively. Division of integers produces an integer result truncated toward zero. Division by zero produces an error message.

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

```
7+-3
```

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

More complex expressions with several operators and with parentheses are interpreted just as in FORTRAN, with exponentiation (^) performed first, then multiplication (*), division (/), modulo (%), and finally, addition (+), and subtraction (-). The contents of parentheses are evaluated before expressions outside the parentheses. All of the above operations are performed from left to right, except exponentiation, which is performed from right to left.

Thus the following two expressions:

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

are equivalent, as are the two expressions:

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

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

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

```
x=x+3
```

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

There is also a built-in square root function whose result is truncated to an integer (see also Section 5.5.3.3, "Scaling"). For example, the lines:

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

produce the printed result:

```
13
```

5.3.2 Specifying Input and Output Bases

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

```
ibase=8
11
```

produce the output line:

```
9
```

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

```
ibase=10
```

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

The statement:

```
ibase=A
```

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

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

```
obase=16
1000
```

produce the output line:

```
3E8
```

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

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

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

5.3.3 Scaling Quantities

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

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

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

Multiplication	The scale of the result is never less than the maximum of the two scales of the operands, never more than the sum of the scales of the operands, and subject to those two restrictions, the scale of the result is set equal to the contents of the internal quantity, **scale.**
Division	The scale of a quotient is the contents of the internal quantity, **scale.**
Modulo	The scale of a remainder is the sum of the scales of the quotient and the divisor.
Exponentiation	The result of an exponentiation is scaled as if the implied multiplications were performed. An exponent must be an integer.
Square Root	The scale of a square root is set to the maximum of the scale of the argument and the contents of **scale.**

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

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

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

```
scale = scale + 1
```

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

```
scale
```

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

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

5.3.4 Using Functions

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

The line:

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

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

The **return** statement can take either of the two forms:

```
return
return(x)
```

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

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

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

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

```
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.

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

If the function "a" is defined as shown above, then the line:

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

would print the result:

```
21.98
```

Similarly, the line:

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

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

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

```
b()
```

calls the function named *b*.

5.3.5 Using Subscripted Variables

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

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

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

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

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

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

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

5.3.6 Using Control Statements: if, while and for

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

> **if** (*relation*) *statement*
> **while** (*relation*) *statement*
> **for** (*expression1* ; *relation* ; *expression2*) *statement*

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.

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

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

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

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

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

which will print the integers from 1 to 10.

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

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

The line:

```
f(a)
```

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

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

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

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

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

5.3.7 Using Other Language Features

Some language features that every user should know about are listed below.

- Normally, statements are entered one to a line. It is also permissible to enter several statements on a line if they are separated by semicolons.

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

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

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

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

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

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

- The following constructions work in **bc** in exactly the same manner as they do in the C language:

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 enter one inadvertently, something legal but unexpected may happen. Be aware that in some of these constructions spaces are significant. There is a real difference between "x=−y" and "x= −y". The first replaces "x" by "x−y" and the second by "−y".

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

- There is a library of math functions that can be obtained by entering:

    bc −l

  when you invoke bc. This command loads the library functions sine, cosine, arctangent, natural logarithm, exponential, and Bessel functions of integer order. These are named "s", "c", "a", "l", "e", and "j(n,x)", respectively. This library sets *scale* to 20 by default.

- If you enter:

    bc *file* ...

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

5.4 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.

5.4.1 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 the characters "/*" and are terminated by "*/".

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

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.

5.4.2 Expressions

All expressions can be evaluated to a value. The value of an expression is always printed unless the main opl 5tor is an assignment. The precedence of expressions (i.e., the order in which they are evaluated) is as follows:

Function calls

Unary operators

Multiplicative operators

Additive operators

Assignment operators

Relational operators

There are several types of expressions:

Named expressions
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 and has an initial value of zero. *Ibase* and *obase* are the input and output number radixes respectively. Both *ibase* and *obase* have initial values of 10.

Constants
Constants are primitive expressions that evaluate to themselves.

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

Function Calls
: Function calls are expressions that return values. They are discussed in section 5.4.3.

5.4.3 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.

5.4.4 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.

5.4.5 Multiplicative Operators

The multiplicative operators (*, /, and %) bind from left to right.

*expr**expr* The result is the product of the two expressions. If "a" and "b" are the scales of the two expressions, then the scale of the result is:

min (a+b, max (scale, a, b))

expr/expr The result is the quotient of the two expressions. The scale of the result is the value of *scale*.

expr%expr The modulo operator (%) produces the remainder of the division of the two expressions. More precisely, *a%b* is *a–a/b*b*. The scale of the result is the sum of the scale of the divisor and the value of *scale*.

expr^expr The exponentiation operator binds right to left. The result is the first expression raised to the power of the second expression. The second expression must be an integer. If "a" is the scale of the left expression and "b" is the absolute value of the right expression, then the scale of the result is:

min (a*b, max (scale, a))

5.4.6 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.

5.4.7 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*.

5.4.8 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

expr>expr

expr<=expr

expr>=expr

expr==expr

expr!=expr

5.4.9 Storage Classes

There are only two storage classes in **bc**: global and automatic (local). Only identifiers that are to be local to a function need 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. They, therefore, do not retain values between function calls. Note that **auto** arrays are specified by the array namer, followed by empty square brackets.

Automatic variables in **bc** do not work the 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.

5.4.10 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. 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 curlybraces ({ and }).

Quoted string statements
: For example:

"*string*"

prints the string inside the quotation marks.

Built-in statements

Built-in statements include **auto, break, define, for, if, quit, return,** and **while.**

The syntax for each built-in statement is given below:

Auto statement

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 statement

The **break** statement causes termination of a **for** or **while** statement. Syntax for the break statement is:

break

Define statement

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 statement

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 statement

The statement is executed if the relation is true. The syntaxis as follows:

> if (*relation*) *statement*

Quit statement

The **quit** statement stops execution of a **bc** program and returns control to XENIX when it is first encountered. Because it is not treated as an executable statement, it cannot be used in a function definition or in an **if, for,** or **while** statement. Note that entering a **Ctrl- d** at the keyboard is the same as entering "quit". The syntax of the quit statement is as follows:

> quit

Return statement

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 as follows:

> return(*expr*)

While statement

The statement is executed while the relation is true. The test occurs before each execution of the statement. The syntax of the while statement is as follows:

> while (*relation*) *statement*

Replace this Page
with Tab Marked:

uucp

Chapter 6

Building a Communication System

6.1 Introduction

This chapter explains how to build a communication system for your computer using either a direct serial line or a normal telephone line and a Hayes Smartmodem. A communication system provides a way to:

- Login to the computer from a remote terminal or computer.
- Use the **cu**(C) command to call and login to other computers.
- Use the **uucp**(C) command to copy files to and from remote computers.
- Use the **uux**(C) command to execute remote commands, including the **mail**(C) program (**rmail**) on a remote computer.

In particular, the communication system is a practical solution to the problem of two micnet networks (see the XENIX *Operations Guide*) that cannot be connected because of distance or cost of cable.

All communication tasks are supported by a variety of files and directories. In addition, the tasks invoked by the **uucp** and **uux** commands are actually performed by a system of underlying programs, called the **uucp** system. The files and underlying programs are described in full later in this chapter.

The following sections explain how to install a modem, and how to configure the necessary files for your system. They also explain how to and maintain a **uucp** system, and describe the details of how it works.

6.1.1 How To Use This Guide

This guide describes how to build a **uucp** system and covers both hardware installation, and software configuration. Topics are first presented in a general form, then in greater detail.

Each control file is first described in general form. Then, you are shown how to configure the file with the **uuinstall** utility.

You do not need to use **uuinstall** to set up your communication system. However, this utility helps you to be more accurate when you configure the system.

Be sure to read the text carefully, since there are many similar commands and actions.

6.2 What You Need

To install a direct wire communication system on your computer, you need

- At least one RS-232 serial line (or serial port) on your computer to use for **uucp** and **cu**.

- The **uucp** programs and files extracted from your XENIX System distribution.

If you want to use your computer as a dial-in and/or dial-out site with a modem, you also need:

- A modem. The default supported modem is a Hayes Smartmodem 1200, but you can modify the supplied dialing routines for other modems.

- A standard telephone jack for access to the telephone system (touch tone line required for the Hayes).

- An RS-232 cable to connect the serial line to the modem.

Finally, since many of the tasks you must perform require special permissions, you must log in to your computer's super-user account before performing them. Check with your computer's system manager before proceeding with this installation, or turn to the XENIX *Operations Guide* for instructions on how to login as the super-user.

6.3 Installing A Direct Wire

This section describes how to install a direct wire between two computers. To connect two computers with a direct wire, you need to:

- Choose a serial line on each machine.

- Connect a serial wire (RS-232) between the two machines, using the chosen serial lines.

- Decide which machine is the dial-in site and which is the dial-out site. The dial-out site calls up and logs in to the dial-in site.

When you finish with these steps, you can proceed with next sections to actually set up the sites.

6.3.1 Choose a Serial Line

On each machine, you must choose the RS-232 serial line you want to use. If there are no lines available, you must install a new serial line or make one available by removing any device connected to it. If you remove a terminal, make sure no one is logged in.

Once you have chosen a serial lines, find the name of the device special file associated with the line by looking in Appendix A of the XENIX *Operations Guide*. The filename should have the form

/dev/tty*nn*

where *nn* is the number of the corresponding line. For example, */dev/tty1a* usually corresponds to serial line 0. You need the name of the actual line for later steps.

The serial line you use for your communication system should be owned by **uucp**. To make sure the line is owned by **uucp** enter this command:

chown uucp /dev/tty*nn*

where *nn* is the number of the corresponding line.

6.3.2 Connect a Serial Wire

You connect two computers together using an RS-232 cable. The actual pin configurations sometimes vary between machines.

Typically, the wire should connect pins 2, 3, and 7 (and/or 20) on one computer to the same pins on the second computer. Sometimes the cable must be *nulled*, which means that pin 2 on one machine is connected to pin 3 on the other, and vice versa.

Since the connections can vary, you should check the hardware manuals for each computer to determine the proper pin connections.

Testing A Connection

For this section, *tty2a* is used as the example serial line for both machines.

To test the wire connection between two machines, follow these steps:

1. Disable the serial lines on each machine. On each computer, enter the command:

   disable /dev/tty2a

2. Attach one end of the serial wire to one of the machines. Attach the other end to the standard data port of a terminal.

3. Enter this command at the computer:

   ```
   (stty 9600; date ) < /dev/tty2a > /dev/tty2a
   ```

   tty2a is our example serial line, and the date command provides sample output.

   You should see the output of the **date** command appear on the terminal screen. Repeat this procedure on the other machine.

   If this doesn't work, check the following:

   - The wire is plugged in properly at each end.
   - The continuity of the wire.
   - The terminal is configured correctly (baud rate, parity, etc.).
   - The serial line is disabled.
   - You are using the correct pin numbers.

Note

An unterminated serial line can cause serious system problems. Do not leave serial lines dangling.

6.4 Installing a Modem

With a modem, you can communicate with computers over standard phone lines. These are the steps to install a modem:

- Choose a serial line.
- Set the dialing configuration.
- Connect the modem.
- Test the connection.

The following sections explain each step in detail. Make sure you inform the telephone company of your intent to use a modem with your telephone line.

You should be particularly careful, since certain telephone services (such as "call waiting") can disrupt **uucp** conversations.

6.4.1 Choose a Serial Line

Choose the RS-232 serial line you want to use with the system and connect to the modem. If there are no lines available, you must install a new serial line or make one available by removing any device connected to it. If you remove a terminal, make sure no one is logged in.

Once you have chosen a serial line, find the name of the device special file associated with the line by looking in Appendix A of the XENIX *Operations Guide.* The filename should have the form

/dev/tty*nn*

where *nn* is the number of the corresponding line. For example, */dev/tty1A* usually corresponds to serial line COMM 1. You need the name of the actual line for later steps.

The serial line you use for your communication system should be owned by *uucp*. To make sure the line is owned by *uucp* enter this command:

chown uucp /dev/tty*nn*

where *nn* is the number of the corresponding line.

6.4.2 Set the Dialing Configuration

In this communication system, your modem can be used to both send and receive calls. You must set the appropriate switches on the modem. (If you are setting up a Hayes Smartmodem 2400, see the next section for configuration instructions.) Follow these steps to configure a Hayes Smartmodem 1200:

1. Remove the front cover of the modem and locate the 8-pin configuration switch. (See the Hayes Reference Manual for instructions on how to remove the cover and locate the switch.)

2. Set the pins on the configuration switch to the following positions:

	1	2	3	4	5	6	7	8
up	●	●		●	●	●	●	
down			●					●

3. Replace the front cover.

If you have a different modem, consult your reference manual for the proper switch settings to both send and receive calls.

6.4.3 Connect the Modem

Once your modem's dialing configuration is set, you are ready to connect the modem to your computer. For proper modem operation, the RS-232 cable must provide the pin connections shown below.

Note that the computer's serial connector must have a DTE (Data Terminal Equipment) configuration. The modem is assumed to have a DCE (Data Communications Equipment) configuration.

Pin Connections

Computer (DTE)	Modem (DCE)
1	1
2	2
3	3
6	6
7	7
8	8
20	20

These pin connections are explained in the *Hayes Reference Manual.*

Review the installation instructions given in the *Hayes Reference Manual,* then follow these steps:

1. Connect the RS-232 serial cable to the serial line connector on the modem, then to the serial line connector on your computer. Make sure the cable is fully connected. (If you are using a Hayes 2400, a 2-3-7 pin cable is not sufficient. We suggest a ribbon cable to connect all appropriate wires.)

2. Plug the telephone line cable into the telephone connector on the modem, then into the telephone wall jack.

3. Plug in the power cord of the modem.

6.4.4 Configuring a Hayes Smartmodem 2400

Although most aspects of modem installation are similar, the Hayes 2400 Smartmodem requires on-line configuration if it is to be used as a dial-in line. Note that the Hayes 2400 will not answer the phone with a 2400 baud carrier if it was not set up with 2400 baud commands.

Make sure that the **/usr/lib/uucp/L-devices** file contains an entry for the line:

```
DIR   ttynn  0      2400
```

You must then configure the modem by issuing set up commands via **cu**(C). Enter:

```
cu -s2400 -l ttynn dir
```

where *nn* is the "tty" number of the serial line. Press RETURN.

Next, enter the following commands to configure the modem. They will be saved in the modem's non-volatile memory. If you do not want to save the settings, do not enter the last command (at&w). Commands are in the left column and short descriptions of what they do are in the right column. Follow each command with a RETURN:

at&f	Fetch factory configuration.
att	Tone dialing.
atl0	Low speaker volume.
at&d2	Set dtr "2": go on hook when dtr drops.
at&c1	Set dcd "1": dcd tracks remote carrier.
ats0=1	Answer phone after 1 ring (AA light should come on).
ats2=128	Disable modem escape sequence.
ate0	No echo (modem will no longer echo what is sent to it).
atq1	Quiet mode (modem will not respond with "OK" after this command or any that follow).
at&w	Saves settings in non-volatile memory.

Exit from **cu** by entering a "tilde" and a "period", followed by a RETURN:

```
~.
```

Set up **dialHA24** as the default dialer program with the following commands:

```
cd /usr/lib/uucp

ln dialHA24 dial
```

The modem is now configured and ready for testing.

6.4.5 Variable Rate Modems

Some modems can determine the connection baud rate from the carrier sent by a remote system. These modems inform the local system of the connection baud rate before issuing the carrier detect signal. The Hayes 2400 dialer supplied with **uucp** detects different connection baud rates and informs **uucp** and **cu** when it exits with a successful connection.

The speed fields in *L-devices* and *L.sys* can specify a range of baud rates for a connection. If a dialer supports baud rates from 300 to 2400 baud, enter the baud rate range in the speed field of *L-devices* as follows:

```
300-2400
```

If a dialer/modem does not allow variable baud rates, place a single baud in the speed field. If a remote system supports several different speeds, place the range of baud rates in the speed field of *L.sys*. If the remote system connects at a single baud rate, place that number in *L.sys*. **uucp** passes the intersection of the *L.sys* and *L-devices* baud rate ranges to the dialer when connecting. If the dialer connects outside of the baud range, it returns a bad baud rate error. Otherwise, it returns the baud rate of the connection.

6.4.6 Test the Modem

As the last step of the modem installation, you should test the modem to make sure that it can send and receive calls. Once you have verified that the modem is working, you can begin to use the communication system.

To test the modem, follow these steps:

1. Start the computer and login as the super-user.

2. Disable the modem serial line by entering

   disable /dev/tty*nn*

   where *nn* is the "tty" number of the serial line.

3. Turn on power to the modem.

4. If you are using a Hayes 1200, make sure the volume switch on the modem is at an appropriate level. You must be able to hear the modem to carry out this test successfully. See the *Hayes Reference Manual* for the location of this switch.

5. Invoke the **dial**(C) program using a command line of the form:

   /usr/lib/uucp/dial /dev/tty*nn* *number speed*

   where /dev/tty*nn* is the filename of your serial line, and *number* is your telephone number (the number of the telephone jack your modem is connected to). **dial** will use **ungetty**(C) to disable the line for the duration of the call. For example, if your serial line is */dev/tty1* and the number is "5551234," enter:

   /usr/lib/uucp/dial /dev/tty1 5551234 1200

6. Listen carefully to the modem. You should hear each digit as the number is dialed, then hear the busy signal when the telephone system tries to make connection with your modem.

7. If the busy signal is present, wait a few moments and listen carefully for the modem to hang up. The modem automatically discontinues any call for which it cannot make a connection.

8. If the busy signal is not present, make sure you have connected the modem to the telephone jack. Make sure the jack is connected to the phone system. Make sure you gave the correct number when invoking **dial**.

9. If you did not hear the modem dial, make sure the volume switch is up. Make sure the modem is connected to the correct serial line and that the cable connection is tight. Make sure you gave the correct filename when invoking **dial**. Make sure modem's power is on.

6.5 Installing a uucp System

When you install the **uucp** system, you configure a series of files which contain information about, and control the actions of the **uucp** programs. You can modify these files with a standard text editor, or you can use the **uuinstall** utility.

To install a **uucp** system you:

- Set up the proper hardware (direct wire or modem).

- Choose a system name for your computer (systemid).
- Create either a dial-in or dial-out site, or a combination of both.
- Create a transmission schedule to ensure that communications operate automatically.

Note that when you install the **uucp** system, or make any modifications, you should be logged in as root. Virtually all of the **uucp** files are writable only by the super-user, and many of them are also readable and executable only by root. Make sure when you are done that all of the **uucp** files are owned by **uucp** and not **root. uucp** will not work correctly if it cannot read or write all of its files.

There are two ways to configure a **uucp** site:

- As a *dial-in* site.
- As a *dial-out* site.

As a dial-in site, other computers call up and log in to your system. They can transfer files and execute certain commands.

As a dial-out site, your computer calls up other computers and logs in. Your computer initiates file transfers to and from the remote machine, as well as local and remote command execution.

Note

The terms *dial-in, dial-out* and *call* describe the communication process for both direct wire and modem/telephone sites.

You can configure your system as both a dial-in and dial-out site, although it cannot function as both at exactly the same time. This is because the serial line is enabled at a dial-in site, and disabled at a dial-out site. Options to configure a port as a dial-in/dial-out line are discussed at the end of "Creating a Dial-out Site."

When you wish to use the port to dial out, the **dial**(M) command uses **ungetty**(C) to disable the port while the call is taking place, and to enable it again when the call is done. Because of this, you should not have to issue a separate command to change the status of the port before making a call.

The following sections explain how to use the **uuinstall** utility to create files for both kinds of sites. They also explain how to create a transmission schedule using **cron** and how to link together micnet sites.

6.5.1 uuinstall

uuinstall helps you install information in various **uucp** control files. To use the **uuinstall** utility, login as root.

Then, enter:

```
# uuinstall
```

The screen displays the main menu:

```
                UUCP Administration Utility

Choose one of the following options:

    1. Examine or update system identification.

    2. Examine or update dial-in or dial-out devices.

    3. Examine or update dialing code abbreviations.

    4. Examine or update system connections.

    5. Examine or update UUCP users.

    6. Terminate this program.

Choose an option:
```

uuinstall returns to this display after performing the action you request. You also return to this main menu if you only press RETURN at a menu option without entering any data.

Each menu option acts on a particular control file:

Menu Option	Control File	Function
1.	/etc/systemid	Contains the **uucp** name of your computer. Other computers on the **uucp** network know your computer by this name.
2.	/usr/lib/uucp/L-devices	Describes the devices on your computer which are connected to other computers on the **uucp** network.
3.	/usr/lib/uucp/L-dialcodes	Contains a list of abbreviations used in the dialing codes for placing calls to other computers.
4.	/usr/lib/uucp/L.sys	Lists the systems on the **uucp** network you can call, or which call you. It also specifies when you can call, or are called, which serial lines are used and the baud rate used.
5.	/usr/lib/uucp/USERFILE	Defines which directories a given site (or a given user) may access using the **uucp** and **uux** commands.

There is an additional file that you cannot change with the **uuinstall** utility:

/usr/lib/uucp/L.cmds	Contains a list of all commands which the **uucp** programs are allowed to execute. This list overrides the default allowable commands. The file can be changed with any standard text editor.

When you have made all necessary changes to the **uucp** control files, enter option '6' to exit from the program. The **uuinstall** utility then displays:

Do you want to update the UUCP control files?

If you enter 'y', the control files are updated. Any other response causes the **uuinstall** utility to exit without making any changes to the files. Each of the options in the main **uuinstall** menu are described in detail later in this chapter.

You can also invoke the **uuinstall** program with a **-r** command line option. This allows you to read the current settings of the **uuinstall** menu options, but will not allow you to make any changes when you exit the program.

You must be the super-user to run **uuinstall**. Refer to **uuinstall**(C) in the XENIX *Reference* for more information.

6.5.2 Choosing a uucp Site Name

In a **uucp** system, every computer belongs to a "site." A site is any computer or any micnet network that can communicate with the **uucp** system.

To distinguish one site from another, every site must have a unique "site name." A site name is any combination of letters and digits that begins with a letter and is no more than seven characters long. The site name may then be used in **uucp** and **uux** commands to direct transmissions to the appropriate computer or micnet network.

The site name should suggest some characteristic of the site, such as its location or affiliation. For example, a site in Chicago can be named "chicago," or a site in the legal department can be named "legal." The site name must be unique. That is, no other computer that calls your computer or is called by your computer can have the same site name.

Once you chose a site name, add it to the */etc/systemid* file as described in the next section.

Creating the systemid File

Each site must have a */etc/systemid* file. The file defines the site name of the given site and associates the site with a micnet network, if any. The file has the form:

sitename
[*machinename*]

sitename — The name of the given site

machinename — The micnet machine name for that computer. If the system is not connected to a micnet network, the *machinename* is optional.

For example, this entry defines a site named "chicago" whose micnet machine name is "brewster":

```
chicago
brewster
```

To set up the *systemid* file for your system, enter:

```
# uuinstall
```

At the main menu, choose option '1.' The current site name and machine name is displayed and you are prompted for any changes to either of these names. If you want to make any changes, you are prompted to enter them.

Since **uucp** systems are often created after a micnet network has been established, the *systemid* file usually already exists on a given site. In this case, you must add the site name to the beginning of each *systemid* file on each computer in the micnet network.

Note that you may list more than one machine name if desired, but each name must be on a separate line. For a full description of the *systemid* file, see **systemid** (M) in the XENIX *Reference Manual*.

6.5.3 Creating a Dial-In Site

To create a dial-in site for logins by remote terminals or computers, you:

- Choose a serial line.
- Disable the serial line.
- Edit the serial line entry in the */etc/ttys* file.

Then you place information into several control files to allow logging in on the serial line you have chosen:

- Use **mkuser**(C) to add a login entry for *uusite*.
- Add user access information to the *USERFILE* file.
- Set access permissions in the *L.cmds* file. Any permissions listed in this file override the **uucp** default permissions.
- Use **uuinstall** to add *uusite* information to *L.sys*.
- Enable the serial line.

Choose a Serial Line

Use a line with modem control for the dial-in line. Refer to "Choose a Serial Line" for more information.

Disable the Serial Line

Disabling the serial line is the next step in creating a dial-in line. To disable the serial line, follow these steps:

1. If you are using a modem, make sure it is installed and tested.

2. Make sure you are logged in as the super-user.

3. Disable the serial line by entering:

   # disable /dev/tty*nn*

   where *nn* is the number of your serial line. If the line is already disabled, the command displays an error message that you can safely ignore.

Edit /etc/ttys

The file */etc/ttys* contains a list of possible login terminals. Enter the command:

cat /etc/ttys

You see a series of entries for the different serial lines.

The form of an entry is:

*xx*tty*nn*

Where:

xx — Two digits. The first is either a one (1), which means the line is enabled, or a zero (0), which means the line is disabled. The second is a number or letter which defines the baud rate of the line.

nn — The number of the tty.

An example entry for a serial line connected to a modem might be:

    02tty2a

The first digit is a zero, which means the line is disabled, so that terminals or computers cannot log in on that line. That digit changes to a one when you use the **enable** command (the next step).

The second digit, '2,' means that the **getty** running on that line cycles the baud rates of that line between 1200, 2400, and 300 baud.

For a direct line, the entry might be:

    06tty2A

If you need to change an entry, do so with a standard text editor. For more information on */etc/ttys*, and the various control codes, see **getty**(M) in the XENIX *Reference*.

Create uucp Login Entries

A dial-in site must provide a login entry for the sites that call it. These entries are placed in the */etc/passwd* file.

A **uucp** login entry has the same form as an ordinary user login entry (see Chapter 3 in the XENIX *Operations Guide*), but has a special login directory and login program instead of the normal user directory and shell.

To create a **uucp** login entry, use the **mkuser**(C) program and follow these steps:

1. Choose a new login name and a user ID for the **uucp** login. The name may be any combination of letters and digits that is no more than eight characters long. The user ID must be an integer number in the range 200 to 65535.

   Make sure the name and ID are unique. A **uucp** login entry must not have the same name or ID as any other login entry.

2. Enter:

   ```
   # /etc/mkuser
   ```

   Follow the program menus and prompts to add the user(s) you wish.

   For a shell type, use "uucp Login."

3. **mkuser** prompts you to enter a password for the new user. This is optional for **uucp** logins.

You can either create new login entries for each site that calls your site, or use one entry for all sites.

Note

A site that dials in to your computer must know if its login has a password. Each site also needs to know what its password is, otherwise it cannot login to your system.

Set Up the USERFILE

The *USERFILE* file defines which directories a given site (or a given user) may access using the **uucp** and **uux** commands. You should create one *USERFILE* entry for each site or user with a login entry in the */etc/passwd* file. Each entry has the form

login,sitename[c] pathname...

login	The login name for a given site (optional).
sitename	The sitename of a given site (optional).
c	The *c* option indicates a callback should occur.
pathname	One or more full pathnames of the directory(s) the given site may access.

The following rules explain how access is granted for each entry.

1. A calling site is granted access to those directories defined in an entry containing its site name.

2. A calling site whose name does not appear in an entry is granted access to the directories defined for the first entry without a site name.

3. A user is granted access to those directories defined in an entry containing his login name.

4. A user whose login name does not appear in an entry is granted access to directories defined in the first entry without a login name.

You may have more than one entry with the same login name if you want. However, you must make sure that at least one of these entries also has the site name of any calling site which can login with that name, or that one of these entries has no site name.

For example, consider the following entries.

```
uuccg,chicago /usr   /usr2/market
uucp,   /usr/vendor
schmidt,   /usr/vendor
,        /usr/spool/uucp/uucppublic
```

The site named "chicago" has access to files in the directories named "/usr" and "/usr2/market". Other sites that login as user "uucp" will be granted access to "/usr/vendor" only. Any local or remote user named "schmidt" is granted access to the directory "/usr/vendor". All other users have access to "/usr/spool/uucp/uucppublic" only.

You can enter this information by choosing option '5' of the **uuinstall** program.

You are prompted as to whether you want to see the current entries in the userfile. If you enter 'y,' you see a screen display similar to the following:

#	Login	Sitename	Paths
0.	uucg	chicago	
1.	ANYLOGIN	ANYSITE	

The ANYLOGIN and ANYSITE entries are special entries displayed whenever a blank login name or site field are encountered in the userfile.

You are then asked whether you want to add or delete an entry in the file. Entries are always deleted by specifying the entry number (#) shown in the first column of the screen display. If you request that an entry be added, you are prompted for the login name, site name and path names of the new entry. If you press RETURN in response to this prompt, the display returns to the main **uuinstall** menu.

In response to the requests for a login name and site name, you may enter the special name "A" (meaning "ANY") which corresponds to a blank field in the userfile for the login or site names. The prompts for pathnames continue until you enter a blank line.

Create the L.cmds File

You do not need to create the file *L.cmds* unless you have special security considerations. The **uucp** login shell may execute only the following default commands:

- lpr
- rmail
- who
- mail
- rnews

If you place a list of commands in the file *L.cmds*, that list supersedes the default allowable commands. If the *L.cmds* file exists, but is empty, no commands are allowed.

If you want your machine to immediately forward files from a remote machine to other machines, the command **/usr/lib/uucp/uucico** must be present in the *L.cmds* file. Otherwise, files will be forwarded the next time your machine connects with the remote machine.

Create the L.sys File

The */usr/lib/uucp/L.sys* file defines the names, telephone numbers, and login information of all sites in the system. (Note that tabs should not be used as field separators.) The file contains one or more entries of the form

sitename time device speed phone login

With:

sitename	The name of the site to be called,
time	A combination of letters and digits that gives the weekdays and times when the given site can be called,
device	The name of the device through which the given site is to be called,
speed	The line speed for the call,
phone	The phone number of the given site, and
login	Login information required to login to the given site.

The *time* defines when the given site can make calls to other sites. It has the form

days times

where *days* is a list of one or more days of the week, and *times* is a range of times of day. The days of the week may be "Su", "Mo", "Tu", "We", "Th", "Fr", "Sa", "Wk", "Any", and "Never". "Wk" means "any weekday," "Any" stands for "any time," and "Never" indicates that the site is never called (except by special request).

The time of day must be given as a four-digit number. The first pair of digits gives the hour (in terms of a 24 hour clock), the second pair gives the minutes. A range of times is a pair of times of the day separated by a hyphen (–). For example, the entry

MoTuTh0800–1230

allows the given site to be called any Monday, Tuesday, or Thursday from 8 in the morning to 12:30 in the afternoon.

The *device* must be the keyword "ACU" if you are using a modem. If you are using a direct line to the other site, then you must give the filename of the serial line (or other device) you intend to use (for example, tty1a).

speed is the baud rate that the remote system will connect at. A range of baud rates may be used here if the remote system can connect at more than one baud rate. A range is specified by a minimum and a maximum baud rate separated by a dash. **uucp** passes the intersection of the *L.sys* speed and *L-devices* speed fields to the dialer.

The *phone* must be the telephone number of the given site. It must have the correct number of digits (including area code if necessary) or be a combination of *L-dialcodes* abbreviations and digits. *L-dialcode* abbreviations must go before any digits. Do not use hyphens. For example, "5551234" is a valid local number and "2065551234" is a valid long distance number. If the abbreviation "sc" is defined to be "555," then "sc" may be used in place of "5551234."

With the Hayes modem, you may use a comma (,) in a number to cause a delay when dialing. This is useful if you must dial for an outside line before placing the call.

For example, the number "9,5551234" causes a delay immediately after the "9" is dialed. After the delay, the rest of the number is dialed. If you are not using a modem, then *phone* must be the filename of the device you intend to use instead of a phone number.

The *login* must be a sequence of names, numbers, and other information that represents the steps required to log in to the given site. This sequence has the form

expect send [*expect send*]...

where *expect* is the prompt or message that you expect the given site to return to the calling site, and *send* is the name, number, or other information that you want to send in response to the expected prompt or message.

For example, the following is the login sequence for a typical XENIX site

ogin: uuccg ssword: market

Note that "ogin:" and "ssword:" are given instead of the complete prompts "Login:" and "Password:". Only the last eight characters in each expected prompt or message are examined, so you do not need to give the preceding characters if you want to save space.

If you anticipate problems during the login sequence, you may include a conditional response immediately after each expected prompt or message. This conditional response has the form

expect [*–send–expect1*] ...

where *expect* is the prompt or message you expect the given site to return, *send* is the name or number you want to send if the prompt or message returned is not correct, and *expect1* is the prompt or message you expect after sending the conditional response. For example, the following shows how to invoke the "login" prompt if it is not immediately present.

--ogin-@-ogin-uuccg ssword: market

There are three special keywords that you may use in the login sequence. The "@" keyword causes an end of transmission character to be sent, the "BREAK" keyword causes a break character to be sent, and the PAUSE*n* keyword causes **uucico** to wait for *n* seconds before continuing.

The complete *L.sys* entry is one line, as shown by the following example:

chicago Any ACU 1200 5551234 ogin: uucp ssword: market

Set up the *L.sys* file by choosing option '4' of the **uuinstall** command. You are prompted as to whether you want to see the current entries in the *L.sys* file.

If you enter 'y,' you see a screen display similar to the following:

```
Entry#:          0
System name:     chicago
Time to call:    Any
Line:            ACU
Speed:           1200
Phone#:          5551234
Login sequence: ogin: uucp ssword: market

Press Enter to see next entry
```

A new entry is displayed each time you press the RETURN key. You are then asked whether you want to add or delete an entry in the file. Entries are always deleted by specifying the entry number (#) shown in the first field for each displayed entry. If you request that an entry be added, you are prompted for each field in turn. If you press RETURN in response to this prompt, you return to the main **uuinstall** menu.

The response to the prompt, concerning the line to use for the call, can be either "A," for an ACU, or the device number of the tty to be used for the connection.

Enable the Serial Line

The next step is to enable the serial line for logins. For example, to enable */dev/tty2A*, enter:

```
# enable tty2A
```

Your computer can now receive calls from remote terminals or computers and prompt for a login name on */dev/tty2A*.

6.5.4 Creating a Dial-Out Site

To create a dial-out site, you choose a serial line and then place information into several control files to allow use of the serial line you have chosen:

- Add user access information to the *USERFILE* file.

- Set access permissions in the *L.cmds* file. Any permissions listed in this file override the **uucp** default permissions.

- Create, or modify, the *L-devices* file.

- Place information about logins on remote computers in the *L.sys*

Then, you install the dialing information your system uses to call and log in to other computers:

- Set up the *dial* program.
- Place dialing abbreviations for remote computers in the *L-dialcodes* file.
- Create a transmission schedule in the form of a shell script to be called periodically by the **cron** program.

With a dial-out line you can call and login to other computers by using the **cu**(C) command. The **cu** command uses the *L-devices* file to locate the correct serial line and set the proper line speed when these values are not explicitly given on the **cu** command line. **cu** also automatically disables the line for the duration of the call with **ungetty**(C).

The following sections explain how to create some of the necessary files. Editing */etc/ttys*, file, creating the *USERFILE*, *L.cmds*, and *L.sys* files and enabling and disabling the lines is discussed in the previous section "Creating a Dial-In Line."

Note

> You can configure a single port to act as a dial-in/dial-out port, with the port toggling from dial-in to dial-out automatically. (Refer to "Dialing In and Out on the Same Line" at the end of this section.)

Set Up the L-devices File

The *L-devices* file defines the devices you intend to use to implement the dial-out line. The file is also used by programs in the **uucp** system (as described later). Use the **uuinstall** utility to set up this file.

Invoke **uuinstall** and select option '2' at the main menu. You are then asked if you want to see the current devices.

Enter 'y' and you see a screen similar to this:

#	Type	Line	Call-Unit	Speed
0.	ACU	tty1a	tty1a	1200
1.	DIR	tty2a		9600

The program prompts if you want to add or delete an entry in the file. Entries are always deleted by specifying the entry number (#) shown in the first column on the screen display.

If you request that an entry be added, you are prompted for the type of unit, either an ACU or a direct line. Enter 'A' for an automatic calling unit (modem) or 'D' for a direct line. You must use capital letters. If you press RETURN in response to this prompt, you are returned to the main menu.

If you specify an ACU, you are prompted for the unit number of the calling unit and the line. Respond with just the number in each case, the **uuinstall** program supplies the "tty" prefix. If you specify a direct line, you are prompted for the line number.

If the *call-unit* field of the *L-devices* file contains a complete pathname, it will be used as the name of the dialer program. The device in the *line* field will be used both as the line and the call unit. This feature allows the use of different modems on different lines, each with a separate dial program.

Finally, you are prompted for the speed of the line. Your response is checked and, if it is invalid, you are prompted for a valid response.

If the modem and dialer connected to this line support variable baud rates, a range of baud rates can be specified by placing the lowest rate and the highest rate, separated by a dash, in the speed field.

uucp will compute the intersection of the speed fields in the *L.sys* and *L-devices* files and pass the result to the dialer when connecting to a remote system. If either *L.sys* or *L-devices* contain a single baud rate, that rate is passed to the dialer which returns an error if the remote system did not connect at that rate.

Set Up the Dial Program

Select (or create) the **dial** program you need for your modem. The default **dial** program is for a Hayes Smartmodem 1200/1200B. Other **dial** programs (for Racal Vadic modems) are also supplied. If you need to use a **dial** program other than the default **dial**, move */usr/lib/uucp/dial* to */usr/lib/uucp/dial.hayes*. You can then move the appropriate **dial** program to */usr/lib/uucp/dial*. The directory */usr/lib/uucp* also includes relinkable files necessary for producing **dial** programs for other kinds of modems. Refer to Chapter 7, "Using Peripheral Devices," in the XENIX *Operations Guide* and the manual pages **dial**(M) and **dial**(S) for more information on creating **dial** programs.

Create the L-dialcodes File

The *L-dialcodes* file defines abbreviations for often used telephone prefixes and area codes. You may use these abbreviations in the *L.sys* file when forming the telephone numbers of remote sites.

The *L-dialcodes* file may contain one or more entries of the form

abbreviation dial-sequence

where *abbreviation* is any combination of letters and digits that begins with a letter, and *dial-sequence* is any combination of digits that represents a telephone prefix, area code or any other part of a telephone number.

For example, the entry

ch 555

defines the abbreviation "ch" to be the telephone prefix "555 ."

Set up the *L-dialcodes* using option '3' of the **uuinstall** program. You are prompted as to whether you want to see the current contents of the L-dialcodes file.

If you enter "y," you see a screen display similar to the following:

#	Abbreviation	Code
0.	Pasa	1818
1.	SntCrz	408
2.	London	011441

You are then prompted to add or delete an entry in the file. Entries are always deleted by specifying the entry number (#), shown in the first column of the screen display. If you request that an entry be added, you are prompted for the abbreviation and the dialing code for each entry. If you press RETURN in response to this prompt, the display returns to the main **uuinstall** menu.

Note entry zero. In order for our example site to call area code 818 (Pasadena, CA), the area code must be prefixed with a one (1). Thus, the dial code "Pasa" is equivalant to "1818."

In the *L.sys* file, here is how you would use this example dial code:

```
plytch Any ACU 1200 Pasa7931211 login uucp ssword: oaktree
```

The next section describes the file in greater detail.

6.5.5 Create a Transmission Schedule

In the **uucp** system, the **uucico** program carries out all transmissions between your site and other sites, sending and receiving files and commands as long as there is work for it to do. On a dial-in site, **uucico** is always started whenever a calling site logs in.

However, on a dial-out site, **uucico** is only started when explicitly invoked. This means you must periodically start up the program on a dial-out site to ensure that all transmissions requested by the **uucp** and **uux** programs are completed.

You can do this in one of two ways:

- Invoke the program manually whenever you need it, or
- Create a shell script and let the **cron** program invoke **uucico** automatically according to a schedule of transmissions.

The most convenient method is to let **cron** run **uucico** for you. To do this, you must choose a schedule of times to invoke **uucico** then create a file */usr/spool/cron/crontabs/uucp* for this schedule. This file has the form:

minutes hour day month day-of-week command-line

where *minutes*, *hour*, *day*, *month*, and *day-of-week* give the exact day of the year and time of day to execute the given *command-line*. Each item, except the *command-line*, must be an integer number within an acceptable range, for example, 0 to 59 for *minutes*.

A sequence of values for one item may be given by separating the values with commas. Also, an asterisk (*) may be given to represent all acceptable values. The *command-line* must be the name of the shell script you have created to invoke **uucico**.

You can add an entry to the */usr/spool/cron/crontabs/uucp* file by using a XENIX text editor. For more information about the file, see **cron** (C) in the XENIX *Reference Manual*. For example, the entry:

```
15,45 * * * * /usr/lib/uucp/transmit
```

invokes the shell script "transmit" every 30 minutes (at 15 minutes past the hour and 45 minutes past the hour) to sites for which requests are pending.

The entry:

```
0 0 * * * /usr/lib/uucp/transmit
```

invokes "transmit" every day at midnight, and the entry:

```
15 2,4,6 * * *  /usr/lib/uucp/transmit
```

invokes the script every day at "2:15," "4:15," and "6:15" in the morning.

A shell script is simply a text file that contains one or more XENIX commands. For example, this shell script automaticalyinvokes *uucico*:

uucico -r1 -s*sitename*

Use the **-s** option if you want to force a call to the given site even if no requests for transmissions exist on the calling site. Note that the -S option may be used in place of the -s option if you want to ignore the range of calling times given in the *L.sys* file. Use one **uucico** command for each site you want to call. If you want to call only those sites for which requests exist, give a single **uucico** command, but do not specify the -s or -S option with the command.

Dialing In and Out on the Same Line

It is possible to use a single port for dial-in and dial-out operations without having to disable/enable it for each use. The dialers distributed with *uucp* perform this function as appropriate when used to dial out to a remote site. All that is necessary is a modification to the *L.sys* file that refers to the line you already configured for your site.

Use the **uuinstall** command. Place the string "Any" in the "time to call" column for the dial-in/dial-out serial line, if it is not already there. The line can now be used as a dial-in/dial-out port.

You can create a shell script by using any XENIX text editor. For convenience, the script should be placed in the */usr/lib/uucp* directory and must be given execute permissions for everyone. Note that you can also add **uucp** maintenance programs to the script. See the section "Creating Maintenance Shell Files" later in this chapter.

6.5.6 Linking micnet Sites

To use a **uucp** system with your micnet network, follow these steps:

1. Add the entry

   uucp:

   to the *maliases* file of the computer on which the **uucp** system is installed.

2. For all other computers in your site, add the entry

   uucp:*machine-name*:

   to the *maliases* file. The *machine-name* must be the name of the computer on which the **uucp** system is installed. This longer form of entry may also be used on the computer on which the **uucp** system is installed.

You can test the **uucp** system by mailing a short letter to yourself via another site. For example, if you are on the site "chicago", and there is another micnet site named "seattle" in the system, then the command

```
mail seattle!chicago!johnd
```

sends mail to the "seattle" site, then back to your "chicago" site, and finally to the user "johnd" in your micnet network. Note that a **uucp** system usually performs its communication tasks according to a fixed schedule and may not return mail immediately.

6.6 Maintaining the System

This section explains how to maintain the uucp system. In particular, it explains how to display and merge the content of uucp log files, how to remove old requests and files from the spool directories, and how to solve some common problems.

You can automate some maintenance tasks by creating shell command files and initiating these files with *crontab* entries. Other tasks require manual modification. Some sample shell files are given toward the end of this section.

6.6.1 Displaying and Merging Log Files

You can display a record of the transmissions requested and completed to a given site or user by using the **uulog**(C) command. The user or sitename must have been previously added to the network with the **uusub -a** command. Any users or sites not added to the network with this command will not be located by **uulog**. The command displays the contents of the individual log files created for a given site or user and merges these entries with the system log file *LOGFILE*. The log files contain information about queued requests, calls to remote sites, execution of **uux** commands, and file copy results. The command has the form

uulog -s*sitename* -u*user*

where -s*sitename* indicates the site whose log files are to be displayed, and -u*user* specifies the user whose log files are to be displayed. If you do not specify a *sitename* and *user*, log files for all sites and users are displayed. The command places the new log files at the beginning of the existing *LOGFILE*.

The log files are originally created in the */usr/spool/uucp* directory as individual files, but should be copied to the *LOGFILE* on a regular basis since they are not copied automatically. For example, the command

uulog

merges all log files and displays their contents. The command

uulog -schicago

merges only log files created for the site "chicago."

Note that the system *LOGFILE* should be removed periodically since it is copied each time new log files are put into the file.

6.6.2 Cleaning the uucp Spool Directory

You can remove unwanted uucp system files from the uucp spool directory by using the **uuclean** command. The command removes temporary data, *LOG*, system status, and lock files from the spool directory if they are more than a given number of hours old. The command has the form

uuclean -d*dir* -m -n*hours* -p*pre* -x*n* where -d*dir* names the directory to be scanned, -*m* causes mail to be sent to the owner of each file removed, -n*hours* gives the age in hours of files to be removed, -p*pre* causes files with the given prefix to be examined and removed, and -x*n* directs the command to give the *n*th level of debugging output. Up to 10 file prefixes may be specified with the **-p** option. If **-m** is used, most mail is sent to the owner of the uucp programs since most files put into the spool directory are owned by the owner of the uucp programs. This is a result of the setuid bit being set on these programs. The default number of *hours* is 72 (3 days).

The **uuclean** program should be run once a day. You can invoke it automatically by using a system daemon such as **cron**. The command

```
uuclean -pTM
```

removes all temporary data files that are at least three days old. The command

```
uuclean -pLCK -h1 -m
```

removes all lock files that are at least an hour old and mails a list of each file removed to the owner.

The **uuclean** command may also be run as needed to remove unwanted files after a system crash or an aborted uucp program.

6.6.3 Reclaiming Log Files after a Crash

You can reclaim individual log files after a system crash by changing their access mode with the **chmod**(C) command, then using **uulog** command. After a transmission failure or system crash, the individual log file for the transmission may be left with access mode 0222 making it impossible for the **uulog** command to read the file. To reclaim the log file, you must use **chmod** to change the access mode to 0666. You can then let **uulog** merge them with the *LOGFILE*.

6.6.4 Reclaiming Data Files after a Crash

You can check the status of files transmitted from a remote site and possibly reclaim some or all of the data lost during an aborted transmission by examining system data files. The data files contain the contents of files copied from remote sites. These files are temporarily kept in the */usr/spool/uucp* directory and their names have the form

TM.*pid.ddd*

where *pid* is a process-id and *ddd* is a sequential three-digit number starting at zero for each invocation of **uucico** and incremented for each file received.

The temporary data files are normally moved to the requested destination immediately after the transmission has finished. However, if a transmission has failed or the system has crashed, the file remains in the spool directory. You can examine the contents of this file with the **cat**(C) command. If desired, you can reclaim the file by moving it to a new location with the mv(C) command. Leftover data files that cannot be reclaimed should be removed using the **uuclean** command.

6.6.5 Checking the Transmission Status

You can check the status of transmissions between sites in the **uucp** system by examining the system status files. System status files contain information about login, dialup, or sequence check failure, as well as the talking status when two machines are conversing.

The files are kept in the */usr/spool/uucp* directory and their names have the form

STST.*sitename*

where *sitename* is the name of the remote site.

Normally, system status files are removed after each successful transmission, but when a failure occurs, the uucp system copies information about the failure to the file and leaves it in the directory. This prevents the uucp system from making further calls to the given site for about an hour, or for sequence check failures, until the file is removed.

To examine the status, use the **cat** command to display the contents of the file. If problems with transmissions are detected it may indicate a problem with the modem or with the serial line connected to the modem.

If a system status file has been left due to a program or system crash, the file may prevent all subsequent transmissions to the given site. In this case, the file must be removed before attempting further calls.

When dialing out a status file will only be created when there is a problem at the remote site. Local modem and line problems do not create status files.

6.6.6 Checking for Locked Sites or Devices

You can make sure the uucp system is not intentionally preventing transmissions to a given site or through a given device by examining the system lock files. The uucp system creates a lock file for each site being called and for each device being used to call a site. Lock files prevent the uucp system from attempting to duplicate conversations with a given site, or from placing multiple calls on the same device.

The lock files are kept in the */usr/spool/uucp* directory and their names have the form

LCK..*str*

where *str* is either a site name or the name of the calling device.

Since lock files prevent all calls to a given site or through a given device, it is wise to make sure no unnecessary lock files are left in the directory. If a transmission has been aborted or the system has crashed, the lock files will prevent subsequent transmissions for about 24 hours. If you want to place a call before this time, you must remove the file using the **uuclean** command.

6.6.7 Creating Maintenance Shell Files

The **uulog** and **uuclean** commands can be invoked automatically by placing them in a shell file and creating a *crontab* file for the shell file. The system daemon **cron** then invokes the commands at the given times and most of the simple maintenance will be performed. For example, you can create a shell file that removes *TM*, *ST*, *LCK* files daily, as well as the *C.* or *D.* files for work which cannot be accomplished for reasons such as bad phone number and login changes. In this case, the shell file should contain the commands

```
/usr/lib/uucp/uuclean -pTM -pC. -pD.
/usr/lib/uucp/uuclean -pST -pLCK -n12
```

Note that the **-n12** option causes the *ST* and *LCK* files older than 12 hours to be deleted. An appropriate *crontab* entry must be created in order to invoke the shell file automatically.

6.7 Details of Operation

This section describes the details of **uucp** system program operation. It explains the processes used to create system communication and defines the files used to support the system.

6.7.1 uucp Programs

The uucp system consists of four primary and four secondary programs. The primary programs are

uucp	This program creates work and gathers data files in the spool directory for the transmission of files.
uux	This program creates work and execute files, and gathers data files for the remote execution of XENIX commands.
uucico	This program executes the work files for data transmission.
uuxqt	This program executes XENIX commands found in execution files.

The secondary programs are

uulog	This program updates the log file with new entries and reports on the status of uucp requests.
uuclean	This program removes old files from the spool directory.
dial	This program directs the modem to dial a remote site.
uusub	This program monitors the uucp network.

6.7.2 uucp Directories and Files

During execution of the **uucp** programs, the **uucp** system uses files from the following three directories:

/usr/lib/uucp
: This is the directory used for **uucp** and **uusub** system files and all executable programs other than **uucp** and **nux.**

/usr/spool/uucp
: This is the spool directory used during **uucp** execution and for the **uusub** *SYSLOG* file.

/usr/spool/uucp/.XQTDIR
: This directory is used during execution of execute files.

Files are created in a spool directory for processing by the uucp daemons. There are three types of files used for the execution of work:

Data files	Contain data for transfer to remote sites

Work files	Contain directions for file transfers between sites
Execution files	Contain directions for XENIX command executions which involve the resources of one or more sites.

6.7.3 uucp – Site-to-Site File Copy

The **uucp** program is the user's primary interface with the system. The **uucp** program was designed to look like the **cp** command. The syntax is

uucp [*option*] ... *source* ... *destination*

where *source* and *destination* may contain the prefix *sitename!* which indicates the site on which the file or files reside or where they will be copied.

Note

uucp makes no distinction between binary and text files. However, the set uid and set gid flags will not accompany the binary file and must be set by someone (or some command) once the binary has arrived at its destination. In addition, the recipient should check the file permissions and ownership for appropriateness.

The options interpreted by **uucp** are

–d	Make directories when necessary for copying the file.
–c	Do not copy source files to the spool directory, but use the specified source when the actual transfer takes place.
–r	Spool only, don't invoke **uucico.**
–m	Send mail on completion of the work.
–n	Notify a user at the remote site that files have arrived. This option will also change the owner of the file to the notified user.

The following options are used primarily for debugging:

–s*dir* Use directory *dir* for the spool directory.

–x*num* Use *num* as the level of debugging output.

The destination may be a directory name, in which case the file name is taken from the last part of the source's name. The source name may contain special shell characters such as "?*[]". If a source argument has a *sitename!* prefix for a remote site, the file name expansion will be done on the remote site.

The command

```
uucp *.c chicago!/usr/dan
```

sets up the transfer of all files whose names end with **.c** to the */usr/dan* directory on the *chicago* machine.

The source and/or destination names may also contain a ~*user* prefix. This translates to the login directory on the specified site. For names with partial pathnames, the current directory is prepended to the file name. File names with "../" are not permitted.

The command

```
uucp chicago!~dan/*.h ~dan
```

sets up the transfer of files whose names end with *.h* in dan's login directory to dan's local login directory.

For each source file, the program checks the source and destination filenames and the site-part of each to classify the work into one of five types:

1. Copy source to destination on local site.

2. Receive files from other sites.

3. Send files to remote sites.

4. Send files from remote sites to another remote site.

5. Receive files from remote sites when the source contains special shell characters as mentioned above.

After the work has been set up in the spool directory, the **uucico** program must be started to try to contact the other machine to execute the work.

Copying Files to a Local Destination

A **cp** command is used to do type 1 work. The **−d** and the **−m** options are not honored in this case.

Receiving Files from Other Sites

For type 2 work a one line work file is created for each file requested, and is put in the spool directory with the following fields, each separated by a blank:

[1] R

[2] The full pathname of the source or a ~user/pathname. The ~*user* part is expanded on the remote site.

[3] The full pathname of the destination file. If the ~*user* notation is used, it will be immediately expanded to be the login directory for the user.

[4] The user's login name.

[5] A "−" followed by an option list. (Only the **−m** and **−d** options appear in this list.)

Sending Files to Remote Sites

For type 3 work, a work file is created for each source file and the source file is copied into a data file in the spool directory. (A **−c** option on the **uucp** program prevents the data file from being made. In this case, the file will be transmitted from the indicated source.) Pathnames are checked using the *USERFILE* to verify access to the requested directory. The fields of each entry are given below.

[1] S

[2] The full pathname of the source file.

[3] The full pathname of the destination or ~user/filename.

[4] The user's login name.

[5] A "−" followed by an option list.

[6] The name of the data file in the spool directory.

[7] The file mode bits of the source file in octal print format (e.g. 0666).

Copying Files Between Sites

For type 4 and 5 work, **uucp** generates a **uucp** command line and sends it to the remote machine; the remote **uucico** executes the command line.

6.7.4 uux – Site To Site Execution

The **uux** command is used to set up the execution of a XENIX command where the execution machine and/or some of the files are remote. The syntax of the **uux** command is

uux [–] [*option*] ... *command-string*

where *command-string* is made up of one or more arguments. All special shell characters such as "<> |" must be quoted either by quoting the entire command string, or by quoting the character as a separate argument. Within the command string, the command and file names may contain a *sitename!* prefix. All arguments which do not contain a "!" are not treated as files. (They are not copied to the execution machine.)

A – (dash) is used to indicate that the standard input for the given command should be inherited from the standard input of the **uux** command. The only option is essentially for debugging: –x*num* directs the command to use a number (1-9) *num* as the level of debugging output. The higher the number, the more debugging information is provided.

The command

pr abc | uux – chicago!rmail joe

will set up the output of "pr abc" as standard input to a **mail** command to be executed on site *usg*.

uux generates an execute file which contains the names of the files required for execution (including standard input), the user's login name, the destination of the standard output, and the command to be executed. This file is either put in the spool directory for local execution or sent to the remote site using a generated send command (type 3 above).

For required files which are not on the execution machine, **uux** will generate receive command files (type 2 above). These command-files are put on the execution machine and executed by the **uucico** program. (This works only if the local site has permission to put files in the remote spool directory as controlled by the remote *USERFILE*.)

The execute file is processed by the **uuxqt** program on the execution machine. It is made up of several lines, each of which contains an identification character and one or more arguments. The order of the lines in the file is not relevant and some of the lines may not be present. Each line is described below.

User Line

U *user site*

where the *user* and *site* are the requestor's login name and site.

Required File Line

F *filename real-name*

where the *filename* is the generated name of a file for the execute machine and *real-name* is the last part of the actual file name (contains no path information). Zero or more of these lines may be present in the execute file. The **uuxqt** program checks for the existence of all required files before the command is executed.

Standard Input Line

I *filename*

The standard input is either specified by a "<" in the command-string or inherited from the standard input of the **uux** command if the **–** option is used. If a standard input is not specified, */dev/null* is used.

Standard Output Line

O *filename sitename*

The standard output is specified by a ">" within the command-string. If a standard output is not specified, */dev/null* is used. (Note that the use of ">>" is not implemented.)

Command Line

C *command* [*arguments*] ...

The *arguments* are those specified in the command string. The standard input and standard output does not appear on this line. All required files are moved to the execution directory (a subdirectory of the spool directory) and the XENIX command is executed using the shell. In addition, a shell PATH statement is prepended to the command line as specified in the **uuxqt** program.

After execution, the standard output is copied or set up to be sent to the proper place.

uux is used by **mail**(C) when sending mail to a remote site. If you do not want **mail** to call the remote site immediately, place the line "spoolonly" in the file */etc/default/mail.*

6.7.5 uucico – Copy In, Copy Out

The **uucico** program performs the following major functions:

- Scan the spool directory for work.
- Place a call to a remote site.
- Negotiate a line protocol to be used.
- Execute all requests from both sites.
- Log work requests and work completions.

uucico may be started by a system daemon, by the user (this is usually for testing), or by a remote site. (The **uucico** program should be specified as the shell field in the */etc/passwd* file for the uucp logins.)

When started with the **–r1** option, the program is considered to be in MASTER mode. In this mode, a connection is made to a remote site. If started by a remote site, the program is considered to be in SLAVE mode.

The MASTER mode operates in one of two ways. If no site name is specified (the **–s** option not specified) the program scans the spool directory for sites to call. If a site name is specified, that site is called, and works only be done for that site.

The **uucico** program must generally be started directly by the user or by another program, such as a shell script invoked by **cron**. There are several options used for execution:

–r1 Start the program in MASTER mode. This is used when **uucico** is started by a program or **cron** shell.

–s*sitename* Do work only for site *sitename*. If **–s** is specified, a call to the specified site is made even if there is no work for site *sitename* in the spool directory, but call only when times in the *L.sys* file permit it. This is useful for polling sites which do not have the hardware to initiate a connection.

–S*sitename* Do work only for site *sitename*. If **–S** is specified, a call to the specified site is made even if there is no work for the site in the site in the spool directory. Unlike **–s**, this option ignores the call times for the *sitename* given in the *L.sys* file.

The following options are used primarily for debugging:

−d*dir* Use directory *dir* for the spool directory.

−x*num* Use *num* as the level of debugging output.

The next part of this section describes the major steps within the **uucico** program.

Scanning For Work

The names of the work related files in the spool directory have the format

type . sitename grade number

where *type* may be "C" for copy command file, "D" for data file, "X" for execute file, *sitename* is the remote site, *grade* is a character, and *number* is a four-digit, padded sequence number.

The file

C.res45n0031

is a work file for a file transfer between the local machine and the "res45" machine.

The scan for work is done by looking through the spool directory for work files (files with prefix "C."). A list is made of all sites to be called. **uucico** calls the site specified by the **−s** or **−S** option and process the corresponding work files.

Calling a Remote Site

The call is made using information from several files which reside in the uucp program directory. At the start of the call process, a lock is set to forbid multiple conversations between the same two sites. The lock filename has the form

LCK..*str*

where *str* is the device name. The file is in the */usr/spool/uucp* directory.

The site name is found in the *L.sys* file. The information contained for each site is

[1] Sitename

[2] Times to call the site (days-of-week and times-of-day)

[3] Device or device type to be used for call

[4] line speed

[5] phone number if field [3] is "ACU," or the device name (same as field [3]) if not

[6] Login information (multiple fields)

The time field is checked against the present time to see if the call should be made.

The *phone number* may contain abbreviations (for example, mh, py, boston) which get translated into dial sequences using the *L-dialcodes* file.

The *L-devices* file is scanned using device type and line speed fields from the *L.sys* file to find an available device for the call. The program trys all devices which satisfy these fields until the call is made or until no more devices can be tried. If a device is successfully opened, a lock file is created so that another copy of **uucico** will not try to use it. If the call is complete, the login information in the last field of *L.sys* is used to login.

The conversation between the two **uucico** programs begins with a handshake started by the SLAVE site. The SLAVE sends a message to let the MASTER know it is ready to receive the site identification and conversation sequence number. The response from the MASTER is verified by the SLAVE and if acceptable, protocol selection begins. The SLAVE can also reply with a call-back required message, in which case the current conversation is terminated.

Select Line Protocol

The remote site sends a message

P*proto-list*

where *proto-list* is a string of characters, each representing a line protocol.

The calling program checks the protocol list for a letter corresponding to an available line protocol and returns a use protocol message. The message has the form

U*code*

where *code* is either a one character protocol letter or "N" which means there is no common protocol.

Processing Work

The initial role of MASTER or SLAVE for the work processing is the mode in which each program starts. (The MASTER has been specified by the **−r1** option.) The MASTER program does a work search similar to the one used in the section "Scanning For Work".

There are five messages used during the work processing, each specified by the first character of the message. They are

S	Send a file
R	Receive a file
C	Copy complete
X	Execute a **uucp** command
H	Hangup

The MASTER sends "R," "S," or "X" messages until all work from the spool directory is complete, at which point an "H" message is sent. The SLAVE replies with the first letter of the request and either the letter "Y" or "N" for yes or no. For example, the message "SY" indicates that it is okay to send a file. While in SLAVE mode, the standard error of **uucico** is redirected to the file */usr/spool/uucp/AUDIT*.

The send and receive replies are based on permission to access the requested file/directory using the *USERFILE* and read/write permissions of the file/directory. After each file is copied into the spool directory of the receiving site, a copy-complete message is sent by the receiver of the file. The message "CY" will be sent if the file has successfully been moved from the temporary spool file to the actual destination. Otherwise, a "CN" message is sent. (In the case of "CN," the transferred file will be in the spool directory with a name beginning with "TM.)" The requests and results are logged on both sites.

The hangup response is determined by the SLAVE program by a work scan of the spool directory. If work for the remote site exists in the SLAVE's spool directory, an "HN" message is sent and the programs switch roles. If no work exists, an "HY" response is sent.

Terminating a Conversation

When an "HY" message is received by the MASTER it is echoed back to the SLAVE and the protocols are turned off. Each program sends a final "OO" message to the other. The original SLAVE program cleans up and terminates. The MASTER proceeds to call other sites and process work as long as possible or terminate if a −s option was specified.

6.7.6 uuxqt – uucp Command Execution

The **uuxqt** program is used to process execute files generated by **uux**. The **uuxqt** program is started by the **uucico** program. The program scans the spool directory for execute files (prefix *X.*). Each one is checked to see if all the required files are available and if so, the command line or send line is executed.

The execute file is described in "uux – Site to Site Execution".

The execution is accomplished by executing the shell command

```
sh -c
```

with the command line after appropriate standard input and standard output have been opened. If a standard output is specified, the program will create a send command or copy the output file as appropriate.

6.7.7 Security

The uucp system, left unrestricted, allows any outside user to execute any commands and copy in/out any file which is readable/writable by the uucp login user. It is up to the individual sites to be aware of this and apply the protections that they feel are necessary.

There are several security features available aside from the normal file mode protections. These must be set up by the installer of the *uucp* system.

The login for uucp does not get a standard shell. Instead, the *uucico* program is started, and receives a special, restricted shell. Therefore, the only work that can be done is through **uucico.**

A path check is done on file names that are to be sent or received. The *USERFILE* supplies the information for these checks. The *USERFILE* can also be set up to require call-back for certain login-ids. See "Uucp Directories and Files."

A conversation sequence count can be set up so that the called system can be more confident of the caller's identity.

The **uuxqt** program comes with a list of commands that it executes. The list of allowable commands can be altered with the *L.cmds* file.

A PATH shell statement is prepended to the command line as specified in the *uuxqt* program.

The *L.sys* file should be owned by uucp and have mode 0400 to protect the phone numbers and login information for remote sites. (The **uucp,**

uucico, uux, and **uuxqt** programs should be also owned by **uucp** and have the setuid bit set.)

When sending files via **uucp** the **-n** *user* option can be used to change the owner and group of the file to *user* on the remote system. Default privileges are read/write all if **-n** is not used.

Replace this Page
with Tab Marked:

C-Shell

Chapter 7

The C-Shell

7.21 Special Characters 7-23

7.1 Introduction

The C-shell program, **csh,** is a command language interpreter for XENIX system users. The C-shell, like the standard XENIX shell *sh*, is an interface between you and the XENIX commands and programs. It translates command lines entered at a terminal into corresponding system actions, gives you access to information, such as your login name, home directory, and mailbox, and lets you construct shell procedures for automating system tasks.

This appendix explains how to use the C-shell. It also explains the syntax and function of C-shell commands and features, and shows how to use these features to create shell procedures. The C-shell is fully described in **csh** (C) in the XENIX *Reference Manual*.

7.2 Invoking the C-shell

You can invoke the C-shell from another shell by using the **csh** command. To invoke the C-shell, enter:

```
csh
```

at the standard shell's command line. You can also direct the system to invoke the C-shell for you when you log in. If you have given the C-shell as your login shell in your */etc/passwd* file entry, the system automatically starts the shell when you login.

After the system starts the C-shell, the shell searches your home directory for the command files *.cshrc* and *.login*. If the shell finds the files, it executes the commands contained in them, then displays the C-shell prompt.

The *.cshrc* file typically contains the commands you wish to execute each time you start a C-shell, and the *.login* file contains the commands you wish to execute afterlogging in to the system. For example, the following is the contents of a typical *.login* file:

```
set ignoreeof
set mail=(/usr/spool/mail/bill)
set time=15
set history=10
mail
```

This file contains several **set** commands. The **set** command is executed directly by the C-shell; there is no corresponding XENIX program for this command. **Set** sets the C-shell variable "ignoreeof" which shields the C-shell from logging out if Ctrl-d is hit. Instead of Ctrl-d, the **logout** command is used to log out of the system. By setting the "mail" variable, the

C-shell is notified that it is to watch for incoming mail and notify you if new mail arrives.

Next the C-shell variable "time" is set to 15 causing the C-shell to automatically print out statistics lines for commands that execute for at least 15 seconds of CPU time. The variable "history" is set to 10 indicating that the C-shell will remember the last 10 commands typed in its history list, (described later).

Finally, the XENIX *mail* program is invoked.

When the C-shell finishes processing the *.login* file, it begins reading commands from the terminal, prompting for each with:

```
%
```

When you logout (by giving the **logout** command) the C-shell prints:

```
logout
```

and executes commands from the file *.logout* if it exists in your home directory. After that, the C-shell terminates and XENIX logs you off the system.

7.3 Using Shell Variables

The C-shell maintains a set of variables. For example, in the above discussion, the variables "history" and "time" had the values 10 and 15. Each C-shell variable has as its value an array of zero or more strings. C-shell variables may be assigned values by the **set** command, which has several forms, the most useful of which is:

```
set name = value
```

C-shell variables may be used to store values that are to be used later in commands through a substitution mechanism. The C-shell variables most commonly referenced are, however, those that the C-shell itself refers to. By changing the values of these variables you can directly affect the behavior of the C-shell.

One of the most important variables is "path". This variable contains a list of directory names. When you enter a command name at your terminal, the C-shell examines each named directory in turn, until it finds an executable file whose name corresponds to the name you entered. The **set** command with no arguments displays the values of all variables currently defined in the C-shell.

The following example shows typical default values:

```
argv   ()
home   /usr/bill
path   (. /bin /usr/bin)
prompt     %
shell  /bin/csh
status 0
```

This output indicates that the variable "path" begins with the current directory indicated by dot (.), then */bin*, and */usr/bin.* Your own local commands may be in the current directory. Normal XENIX commands reside in */bin* and */usr/bin.*

Sometimes a number of locally developed programs reside in the directory */usr/local.* If you want all C-shells that you invoke to have access to these new programs, place the command:

```
set path=(. /bin /usr/bin /usr/local)
```

in the *.cshrc* file in your home directory. Try doing this, then logging out and back in. Enter:

```
set
```

to see that the value assigned to "path" has changed.

You should be aware that when you log in the C-shell examines each directory that you insert into your path and determines which commands are contained there, except for the current directory which the C-shell treats specially. This means that if commands are added to a directory in your search path after you have started the C-shell, they will not necessarily be found. If you wish to use a command which has been added after you have logged in, you should give the command:

```
rehash
```

to the C-shell. **Rehash** causes the shell to recompute its internal table of command locations, so that it will find the newly added command. Since the C-shell has to look in the current directory on each command anyway, placing it at the end of the path specification usually works best and reduces overhead.

Other useful built in variables are "home" which shows your home directory, and "ignoreeof" which can be set in your *.login* file to tell the C-shell not to exit when it receives an end-of-file from a terminal. The variable "ignoreeof" is one of several variables whose value the C-shell does not care about; the C-shell is only concerned with whether these variables are set or unset.

Thus, to set "ignoreeof" you simply enter:

```
set ignoreeof
```

and to unset it enter:

```
unset ignoreeof
```

Some other useful built-in C-shell variables are "noclobber" and "mail". The syntax:

```
>filename
```

which redirects the standard output of a command just as in the regular shell, overwrites and destroys the previous contents of the named file. In this way, you may accidentally overwrite a file which is valuable. If you prefer that the C-shell not overwrite files in this way you can:

```
set noclobber
```

in your *.login* file. Then entering:

```
date > now
```

causes an error message if the file *now* already exists. You can enter:

```
date >! now
```

if you really want to overwrite the contents of *now*. The ">!" is a special syntax indicating that overwriting or "clobbering" the file is ok. (The space between the exclamation point (!) and the word "now" is critical here, as "!now" would be an invocation of the history mechanism, described below, and have a totally different effect.)

7.4 Using the C-Shell History List

The C-shell can maintain a history list into which it places the text of previous commands. It is possible to use a notation that reuses commands, or words from commands, in forming new commands. This mechanism can be used to repeat previous commands or to correct minor typing mistakes in commands.

The following figure gives a sample session involving typical usage of the history mechanism of the C-shell. Boldface indicates user input:

```
% cat bug.c
main()
{
        printf("hello);
}
% cc !$
cc bug.c
bug.c(4) :error 1: newline in constant
%  ed !$
ed bug.c
28
3s/);/"&/p
        printf("hello");
w
29
q
% !c
cc bug.c
% a.out
hello% !e
ed bug.c
29
3s/lo/lo\\n/p
        printf("hello\n");
w
31
q
% !c -o bug
cc bug.c -o bug
% size a.out bug
a.out: 5124 + 614 + 1254 = 6692 = 0x1b50
bug: 5124 + 616 + 1252 = 6692 = 0x1b50
% ls -l !*
ls -l a.out bug
-rwxr-xr-x 1 bill        7648 Dec  19 09:41 a.out
-rwxr-xr-x 1 bill        7650 Dec  19 09:42 bug
% bug
hello
% pr bug.c | lpt
lpt: Command not found.
% ^lpt^lpr
pr bug.c | lpr
%
```

In this example, we have a very simple C program that has a bug or two in the file *bug.c*, which we **cat** out on our terminal. We then try to run the C compiler on it, referring to the file again as "!$", meaning the last argument to the previous command. Here the exclamation mark (!) is the history mechanism invocation metacharacter, and the dollar sign ($) stands for the last argument, by analogy to the dollar sign in the editor which stands for the end-of-line.

The C-shell echoed the command, as it would have been typed without use of the history mechanism, and then executed the command. The compilation yielded error diagnostics, so we now edit the file we were trying to compile, fix the bug, and run the C compiler again, this time referring to this command simply as "!c", which repeats the last command that started with the letter "c".

If there were other commands beginning with the letter "c" executed recently, we could have said "!cc" or even "!cc:p" which prints the last command starting with "cc" without executing it, so that you can check to see whether you really want to execute a given command.

After this recompilation, we ran the resulting *a.out* file, and then noting that there still was a bug, ran the editor again. After fixing the program we ran the C compiler again, but tacked onto the command an extra "–o bug" telling the compiler to place the resultant binary in the file *bug* rather than *a.out*. In general, the history mechanisms may be used anywhere in the formation of new commands, and other characters may be placed before and after the substituted commands.

We then ran the **size** command to see how large the binary program images we have created were, and then we ran an "ls –l" command with the same argument list, denoting the argument list:

```
!*
```

Finally, we ran the program *bug* to see that its output is indeed correct.

To make a listing of the program, we ran the **pr** command on the file *bug.c*. In order to print the listing at a lineprinter we piped the output to **lpr**, but misspelled it as "lpt". To correct this we used a C-shell substitute, placing the old text and new text between caret (^) characters. This is similar to the substitute command in the editor. Finally, we repeated the same command with:

```
!!
```

and sent its output to the lineprinter.

There are other mechanisms available for repeating commands. The **history** command prints out a numbered list of previous commands. You can then refer to these commands by number. There is a way to refer to a previous command by searching for a string which appeared in it, and there are other, less useful, ways to select arguments to include in a new command. A complete description of all these mechanisms is given in **csh** (C) the XENIX *Reference Manual.*

7.5 Using Aliases

The C-shell has an alias mechanism that can be used to make transformations on commands immediately after they are input. This mechanism can be used to simplify the commands you enter, to supply default arguments to commands, or to perform transformations on commands and their arguments. The alias facility is similar to a macro facility. Some of the features obtained by aliasing can be obtained also using C-shell command files, but these take place in another instance of the C-shell and cannot directly affect the current C-shell's environment or involve commands such as **cd** which must be done in the current C-shell.

For example, suppose there is a new version of the mail program on the system called *newmail* that you wish to use instead of the standard mail program *mail*. If you place the C-shell command

```
alias mail newmail
```

in your *.cshrc* file, the C-shell will transform an input line of the form:

```
mail bill
```

into a call on *newmail*. Suppose you wish the command **ls** to always show sizes of files, that is, to always use the **–s** option. In this case, you can use the **alias** command to do:

```
alias ls ls –s
```

or even:

```
alias dir ls –s
```

creating a new command named **dir**. If we then enter:

```
dir ~bill
```

the C-shell translates this to:

```
ls –s /usr/bill
```

Note that the tilde (~) is a special C-shell symbol that represents the user's home directory.

Thus the **alias** command can be used to provide short names for commands, to provide default arguments, and to define new short commands in terms of other commands. It is also possible to define aliases that contain multiple commands or pipelines, showing where the arguments to the original command are to be substituted using the facilities of the history mechanism.

Thus the definition:

```
alias cd 'cd \!* ; ls '
```

specifies an **ls** command after each **cd** command. We enclosed the entire alias definition in single quotation marks (') to prevent most substitutions from occurring and to prevent the semicolon (;) from being recognized as a metacharacter. The exclamation mark (!) is escaped with a backslash (\) to prevent it from being interpreted when the alias command is entered. The "\!*" here substitutes the entire argument list to the prealiasing **cd** command; no error is given if there are no arguments. The semicolon separating commands is used here to indicate that one command is to be done and then the next. Similarly the following example defines a command that looks up its first argument in the password file.

```
alias whois 'grep \!^ /etc/passwd'
```

The C-shell currently reads the *.cshrc* file each time it starts up. If you place a large number of aliases there, C-shells will tend to start slowly. You should try to limit the number of aliases you have to a reasonable number (10 or 15 is reasonable). Too many aliases causes delays and makes the system seem sluggish when you execute commands from within an editor or other programs.

7.6 Redirecting Input and Output

In addition to the standard output, commands also have a diagnostic output that is normally directed to the terminal even when the standard output is redirected to a file or a pipe. It is occasionally useful to direct the diagnostic output along with the standard output. For instance, if you want to redirect the output of a long running command into a file and wish to have a record of any error diagnostic it produces you can enter:

```
command >& file
```

The ">&" here tells the C-shell to route both the diagnostic output and the standard output into *file*. Similarly you can give the command:

```
command |& lpr
```

to route both standard and diagnostic output through the pipe to the line-printer. The form:

```
command >&! file
```

is used when "noclobber" is set and *file* already exists.

Finally, use the form:

```
command >> file
```

to append output to the end of an existing file. If "noclobber" is set, then an error results if *file* does not exist, otherwise the C-shell creates *file*. The form:

```
command >>! file
```

lets you append to a file even if it does not exist and "noclobber" is set.

7.7 Creating Background and Foreground Jobs

When one or more commands are entered together as a pipeline or as a sequence of commands separated by semicolons, a single job is created by the C-shell consisting of these commands together as a unit. Single commands without pipes or semicolons create the simplest jobs. Usually, every line entered to the C-shell creates a job. Each of the following lines creates a job:

```
sort < data
ls -s | sort -n | head -5
mail harold
```

If the ampersand metacharacter (&) is entered at the end of the commands, then the job is started as a background job. This means that the C-shell does not wait for the job to finish, but instead, immediately prompts for another command. The job runs in the background at the same time that normal jobs, called foreground jobs, continue to be read and executed by the C-shell. Thus:

```
du > usage &
```

runs the *du* program, which reports on the disk usage of your working directory, puts the output into the file *usage* and returns immediately with a prompt for the next command without waiting for *du* to finish. The *du* program continues executing in the background until it finishes, even though you can enter and execute more commands in the mean time. Background jobs are unaffected by any signals from the keyboard such as the **INTERRUPT** or **QUIT** signals.

The **kill** command terminates a background job immediately. Normally, this is done by specifying the process number of the job you want killed. Process numbers can be found with the **ps** command.

7.8 Using Built-In Commands

This section explains how to use some of the built-in C-shell commands.

The **alias** command described above is used to assign new aliases and to display existing aliases. If given no arguments, **alias** prints the list of current aliases. It may also be given one argument, such as to show the current alias for a given string of characters. For example:

```
alias ls
```

prints the current alias for the string "ls".

The **history** command displays the contents of the history list. The numbers given with the history events can be used to reference previous events that are difficult to reference contextually. There is also a C-shell variable named "prompt". By placing an exclamation point (!) in its value the C-shell will substitute the number of the current command in the history list. You can use this number to refer to a command in a history substitution. For example, you could enter:

```
set prompt='\! % '
```

Note that the exclamation mark (!) had to be escaped here even within back quotes.

The **logout** command is used to terminate a login C-shell that has "ignoreeof" set.

The **rehash** command causes the C-shell to recompute a table of command locations. This is necessary if you add a command to a directory in the current C-shell's search path and want the C-shell to find it, since otherwise the hashing algorithm may tell the C-shell that the command wasn't in that directory when the hash table was computed.

The **repeat** command is used to repeat a command several times. Thus to make 5 copies of the file *one* in the file *five* you could enter:

```
repeat 5 cat one >> five
```

The **setenv** command can be used to set variables in the environment. Thus:

```
setenv TERM adm3a
```

sets the value of the environment variable "TERM" to "adm3a". The program *env* exists to print out the environment. For example, its output might look like this:

```
HOME=/usr/bill
SHELL=/bin/csh
PATH=:/usr/ucb:/bin:/usr/bin:/usr/local
TERM=adm3a
USER=bill
```

The **source** command is used to force the current C-shell to read commands from a file. Thus:

```
source .cshrc
```

can be used after editing in a change to the *.cshrc* file that you wish to take effect before the next time you login.

The **time** command is used to cause a command to be timed no matter how much CPU time it takes. Thus:

```
time cp /etc/rc /usr/bill/rc
```

displays:

```
0.0u 0.1s 0:01 8%
```

Similarly:

```
time wc /etc/rc /usr/bill/rc
```

displays:

```
      52     178    1347 /etc/rc
      52     178    1347 /usr/bill/rc
     104     356    2694 total
 0.1u 0.1s 0:00 13%
```

This indicates that the **cp** command used a negligible amount of user time (u) and about 1/10th of a second system time (s); the elapsed time was 1 second (0:01). The word count command **wc** used 0.1 seconds of user time and 0.1 seconds of system time in less than a second of elapsed time. The percentage "13%" indicates that over the period when it was active the **wc** command used an average of 13 percent of the available CPU cycles of the machine.

The **unalias** and **unset** commands are used to remove aliases and variable definitions from the C-shell. The command **unsetenv** removes variables from the environment.

7.9 Creating Command Scripts

It is possible to place commands in files and to cause C-shells to be invoked to read and execute commands from these files, which are called C-shell scripts. This section describes the C-shell features that are useful when creating C-shell scripts.

7.10 Using the argv Variable

A **csh** command script may be interpreted by saying:

```
csh script argument ...
```

where *script* is the name of the file containing a group of C-shell commands and *argument* is a sequence of command arguments. The C-shell places these arguments in the variable "argv" and then begins to read commands from *script*. These parameters are then available through the same mechanisms that are used to reference any other C-shell variables.

If you make the file *script* executable by doing:

```
chmod 755 script
```

or:

```
chmod +x script
```

and then place a C-shell comment at the beginning of the C-shell script (i.e., begin the file with a number sign (#)) then */bin/csh* will automatically be invoked to execute *script* when you enter:

```
script
```

If the file does not begin with a number sign (#) then the standard shell */bin/sh* will be used to execute it.

7.11 Substituting Shell Variables

After each input line is broken into words and history substitutions are done on it, the input line is parsed into distinct commands. Before each command is executed a mechanism known as variable substitution is performed on these words. Keyed by the dollar sign ($), this substitution replaces the names of variables by their values. Thus:

```
echo $argv
```

when placed in a command script would cause the current value of the variable "argv" to be echoed to the output of the C-shell script. It is an error for "argv" to be unset at this point.

A number of notations are provided for accessing components and attributes of variables. The notation:

$?*name*

expands to 1 if *name* is set or to 0 if *name* is not set. It is the fundamental mechanism used for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The notation:

```
$#name
```

expands to the number of elements in the variable "name". To illustrate, examine the following terminal session (input is in boldface):

```
% set argv=(a b c)
% echo $?argv
1
% echo $#argv
3
% unset argv
% echo $?argv
0
% echo $argv
Undefined variable: argv.
%
```

It is also possible to access the components of a variable that has several values. Thus:

```
$argv[1]
```

gives the first component of "argv" or in the example above "a". Similarly:

```
$argv[$#argv]
```

would give "c", and:

```
$argv[1-2]
```

would give:

```
a b
```

Other notations useful in C-shell scripts are:

$*n*

where *n* is an integer. This is shorthand for:

$argv[*n*]

the *n*'th parameter and:

```
$*
```

which is a shorthand for:

```
$argv
```

The form:

```
$$
```

expands to the process number of the current C-shell. Since this process number is unique in the system, it is often used in the generation of unique temporary filenames.

One minor difference between "$*n*" and "$argv[*n*]" should be noted here. The form: "$argv[*n*]" will yield an error if *n* is not in the range 1–$#argv while "$n" will never yield an out-of-range subscript error. This is for compatibility with the way older shells handle parameters.

Another important point is that it is never an error to give a subrange of the form: "n–"; if there are less than "n" components of the given variable then no words are substituted. A range of the form: "m–n" likewise returns an empty vector without giving an error when "m" exceeds the number of elements of the given variable, provided the subscript "n" is in range.

7.12 Using Expressions

To construct useful C-shell scripts, the C-shell must be able to evaluate expressions based on the values of variables. In fact, all the arithmetic operations of the C language are available in the C-shell with the same precedence that they have in C. In particular, the operations "==" and "!=" compare strings and the operators "&&" and "||" implement the logical AND and OR operations. The special operators "=~" and "!~" are similar to "==" and "!=" except that the string on the right side can have pattern matching characters (like *, ? or [and]). These operators test whether the string on the left matches the pattern on the right.

The C-shell also allows file inquiries of the form:

–? *filename*

where question mark (?) is replaced by a number of single characters. For example, the expression primitive:

–e *filename*

tells whether *filename* exists. Other primitives test for read, write and execute access to the file, whether it is a directory, or if it has nonzero length.

It is possible to test whether a command terminates normally, by using a primitive of the form:

{ *command* }

which returns 1 if the command exits normally with exit status 0, or 0 if the command terminates abnormally or with exit status nonzero. If more detailed information about the execution status of a command is required, it can be executed and the "status" variable examined in the next command. Since "$status" is set by every command, its value is always changing.

For the full list of expression components, see **csh**(C) in the XENIX *Reference Manual.*

7.13 Using the C-Shell: A Sample Script

A sample C-shell script follows that uses the expression mechanism of the C-shell and some of its control structures:

```
#
# Copyc copies those C programs in the specified list
# to the directory ~/backup if they differ from the files
# already in ~/backup
#
set noglob
foreach i ($argv)

        if ($i !~ *.c) continue  # not a .c file so do nothing

        if (! -r ~/backup/$i:t) then
                echo $i:t not in backup... not cp\'ed
                continue
        endif

        cmp -s $i ~/backup/$i:t # to set $status

        if ($status != 0) then
                echo new backup of $i
                cp $i ~/backup/$i:t
        endif
end
```

This script uses the **foreach** command, which iteratively executes the group of commands between the **foreach** and the matching **end** statements for each valie value of the variable "i". If you want to look more closely at what happends during execution of a **foreach** loop, you can use the debug command **break** to stop execution at any point and the debug command **continue** to resume execution. The value of the iteration variable (*i* in this case) will stay at whatever it was when the last **foreach** loop was completed.

The "noglob" variable is set to prevent filename expansion of the members of "argv". This is a good idea, in general, if the arguments to a C-shell script are filenames which have already been expanded or if the arguments may contain filename expansion metacharacters. It is also possible to quote each use of a "$" variable expansion, but this is harder and less reliable.

The other control construct is a statement of the form:

```
if ( expression ) then
        command
        ...
endif
```

The placement of the keywords in this statement is not flexible due to the current implementation of the C-shell. The following two formats are not acceptable to the C-shell:

```
if ( expression ) # Won't work!
then
        command
        ...
endif
```

and:

```
if (expression) then command endif # Won't work
```

The C-shell does have another form of the if statement:

```
if ( expression ) command
```

which can be written:

```
if ( expression ) \
        command
```

Here we have escaped the newline for the sake of appearance. The command must not involve " |", "&" or ";" and must not be another control command. The second form requires the final backslash (\) to immediately precede the end-of-line.

The more general **if** statements above also admit a sequence of **else–if** pairs followed by a single **else** and an **endif**, for example:

```
if ( expression ) then
        commands
else if ( expression ) then
        commands
...

else
        commands
endif
```

Another important mechanism used in C-shell scripts is the colon (:) modifier. We can use the modifier **:r** here to extract the root of a filename or **:e** to extract the extension. Thus if the variable "i" has the value */mnt/foo.bar*then

```
echo $i $i:r $i:e
```

produces:

```
/mnt/foo.bar /mnt/foo bar
```

This example shows how the **:r** modifier strips off the trailing ".bar" and the **:e** modifier leaves only the "bar". Other modifiers take off the last component of a pathname leaving the head **:h** or all but the last component of a pathname leaving the tail **:t**. These modifiers are fully described in the **csh**(C) entry in the XENIX *Reference Manual.* It is also possible to use the command substitution mechanism to perform modifications on strings to then reenter the C-shell environment. Since each usage of this mechanism involves the creation of a new process, it is much more expensive to use than the colon (:) modification mechanism. It is also important to note that the current implementation of the C-shell limits the number of colon modifiers on a "$" substitution to 1. Thus:

```
% echo $i $i:h:t
```

produces:

```
/a/b/c /a/b:t
```

and does not do what you might expect.

Finally, we note that the number sign character (#) lexically introduces a C-shell comment in C-shell scripts (but not from the terminal). All subsequent characters on the input line after a number sign are discarded by the C-shell. This character can be quoted using "'" or "\" to place it in an argument word.

7.14 Using Other Control Structures

The C-shell also has control structures **while** and **switch** similar to those of C. These take the forms:

```
while ( expression )
        commands
end
```

and:

```
switch (word )

case str1:
        commands
        breaksw

...

case strn:
        commands
        breaksw

default:
        commands
        breaksw

endsw
```

For details see the manual section for **csh**(C). C programmers should note that we use **breaksw** to exit from a **switch** while **break** exits a **while** or **foreach** loop. A common mistake to make in C-shell scripts is to use **break** rather than **breaksw** in switches.

Finally, the C-shell allows a **goto** statement, with labels looking like they do in C:

```
loop:
        commands
        goto loop
```

7.15 Supplying Inputto Commands

Commands run from C-shell scripts receive by default the standard input of the C-shell which is running the script. It allows C-shell scripts to fully participate in pipelines, but mandates extra notation for commands that are to take inline data.

Thus we need a metanotation for supplying inline data to commands in C-shell scripts. For example, consider this script which runs the editor to delete leading blanks from the lines in each argument file:

```
# deblank -- remove leading blanks
foreach i ($argv)
ed - $i << ´ EOF´
1,$s/^[ ]*//
w
q
´EOF´
end
```

The notation:

```
<< ´EOF´
```

means that the standard input for the **ed** command is to come from the text in the C-shell script file up to the next line consisting of exactly EOF. The fact that the EOF is enclosed in single quotation marks (´), i.e., it is quoted, causes the C-shell to not perform variable substitution on the intervening lines. In general, if any part of the word following the "<<" which the C-shell uses to terminate the text to be given to the command is quoted then these substitutions will not be performed. In this case since we used the form "1,$" in our editor script we needed to insure that this dollar sign was not variable substituted. We could also have insured this by preceding the dollar sign ($) with a backslash (\ ), i.e.:

```
1,\$s/^[ ]*//
```

Quoting the EOF terminator is a more reliable way of achieving the same thing.

7.16 Catching Interrupts

If our C-shell script creates temporary files, we may wish to catch interruptions of the C-shell script so that we can clean up these files. We can then do:

```
onintr label
```

where *label* is a label in our program. If an interrupt is received the C-shell will do a "goto label" and we can remove the temporary files, then do an **exit** command (which is built in to the C-shell) to exit from the C-shell script. If we wish to exit with nonzero status we can write:

```
exit(1)
```

to exit with status 1.

7.17 Using Other Features

There are other features of the C-shell useful to writers of C-shell procedures. The **verbose** and **echo** options and the related **−v** and **−x** command line options can be used to help trace the actions of the C-shell. The **−n** option causes the C-shell only to read commands and not to execute them and may sometimes be of use.

One other thing to note is that the C-shell will not execute C-shell scripts that do not begin with the number sign character (#), that is C-shell scripts that do not begin with a comment.

There is also another quotation mechanism using the double quotation mark ("), which allows only some of the expansion mechanisms we have so far discussed to occur on the quoted string and serves to make this string into a single word as the single quote (´) does.

7.18 Starting a Loop at a Terminal

It is occasionally useful to use the **foreach** control structure at the terminal to aid in performing a number of similar commands. For instance, if there were three shells in use on a particular system, */bin/sh*, */bin/nsh*, and */bin/csh*, you could count the number of persons using each shell by using the following commands:

```
grep -c csh$ /etc/passwd
grep -c nsh$ /etc/passwd
grep -c -v sh$ /etc/passwd
```

Since these commands are very similar we can use **foreach** to simplify them:

```
$ foreach i ('sh$' 'csh$' '-v sh$')
? grep -c $i /etc/passwd
? end
```

Note here that the C-shell prompts for input with "? " when reading the body of the loop. This occurs only when the **foreach** command is entered interactively.

Also useful with loops are variables that contain lists of filenames or other words. For example, examine the following terminal session:

```
% set a=(`ls`)
% echo $a
csh.n csh.rm/fR
% ls
csh.n
csh.rm
% echo $#a
2
```

The **set** command here gave the variable "a" a list of all the filenames in the current directory as value. We can then iterate over these names to perform any chosen function.

The output of a command within back quotation marks (`) is converted by the C-shell to a list of words. You can also place the quoted string within double quotation marks (") to take each (nonempty) line as a component of the variable. This prevents the lines from being split into words at blanks and tabs. A modifier **:x** exists which can be used later to expand each component of the variable into another variable by splitting the original variable into separate words at embedded blanks and tabs.

7.19 Using Braces with Arguments

Another form of filename expansion involves the characters, "{" and "}". These characters specify that the contained strings, separated by commas (,) are to be consecutively substituted into the containing characters and the results expanded left to right. Thus:

```
A{str1,str2,...strn}B
```

expands to:

```
Astr1B Astr2B ... AstrnB
```

This expansion occurs before the other filename expansions, and may be applied recursively (i.e., nested). The results of each expanded string are sorted separately, left to right order being preserved. The resulting filenames are not required to exist if no other expansion mechanisms are used. This means that this mechanism can be used to generate arguments which are not filenames, but which have common parts.

A typical use of this would be:

```
mkdir ~/{hdrs,retrofit,csh}
```

to make subdirectories *hdrs*, *retrofit* and *csh* in your home directory. This mechanism is most useful when the common prefix is longer than in this example:

```
chown root /usr/demo/{file1,file2,...}
```

7.20 Substituting Commands

A command enclosed in accent symbols (`) is replaced, just before filenames are expanded, by the output from that command. Thus, it is possible to do:

```
set pwd=`pwd`
```

to save the current directory in the variable "pwd" or to do:

```
vi `grep -l TRACE *.c`
```

to run the editor **vi** supplying as arguments those files whose names end in which have the string "TRACE" in them. Command expansion also occurs in input redirected with "<<" and within quotation marks ("). Refer to csh(C) in the XENIX *Reference Manual* for more information.

7.21 Special Characters

The following table lists the special characters of **csh** and the XENIX system. A number of these characters also have special meaning in expressions. See the **csh** manual section for a complete list.

Syntactic metacharacters

;	Separates commands to be executed sequentially
|	Separates commands in a pipeline
( )	Brackets expressions and variable values
&	Follows commands to be executed without waiting for completion

Filename metacharacters

/	Separates components of a file's pathname
.	Separates root parts of a filename from extensions

?	Expansion character matching any single character
*	Expansion character matching any sequence of characters
[ ]	Expansion sequence matching any single character from a set of characters
~	Used at the beginning of a filename to indicate home directories
{ }	Used to specify groups of arguments with common parts

Quotation metacharacters

\	Prevents meta-meaning of following single character
'	Prevents meta-meaning of a group of characters
"	Like', but allows variable and command expansion

Input/output metacharacters

<	Indicates redirected input
>	Indicates redirected output

Expansion/Substitution Metacharacters

$	Indicates variable substitution
!	Indicates history substitution
:	Precedes substitution modifiers
^	Used in special forms of history substitution
`	Indicates command substitution

Other Metacharacters

#	Begins scratch filenames; indicates C-shell comments
–	Prefixes option (flag) arguments to commands

Replace this Page
with Tab Marked:

Visual Shell

Chapter 8

Using The Visual Shell

8.1 What is the Visual Shell?

The visual shell, **vsh**, is a menu-driven XENIX shell. This chapter describes the use and behavior of the vsh. This chapter assumes that the reader is familiar with some general XENIX concepts, specifically the structure of XENIX filesystems and the nature of a XENIX 'command'. No familiarity with any other shell, however, is assumed. If you are a first-time user of the visual shell, please completely read the narrative sections of this chapter.

A 'shell' is a program which passes a command to an operating system, and displays the result of running the command. The XENIX shells can also create 'pipelines' for passing the output of one command to another command or 'redirect' the output into a file.

The other XENIX shells available are **sh** and **csh.** These shells are called 'command-line oriented' shells. This means that the user enters commands one line at a time. The **sh** and **csh** shells are full computer languages which require study and some programming knowledge to use effectively. These command-line shells are powerful and efficient.

The vsh is a 'menu-oriented' shell. In a menu-oriented shell, the user is given the available commands, or some of the available commands. The user can run the command, by selecting from the menu.

The visual shell is a good shell for users who may not want to master a programming language right away just to use XENIX or a specfic XENIX application. All visual shell users should additionally become familiar with some command-line shell usage.

Users familiar with command-line shells are in for a pleasant surprise if they try the visual shell. Experienced users will appreciate the efficiency and versatility of the visual shell. The distinction is very much akin to the difference between a line-oriented text editor and a full-screen editor.

A menu shell can be used effectively with very little study. On the other hand, a menu shell can also restrict the user from using the operating system in creative, possibly more efficient ways. The Microsoft visual shell strikes a balance in this regard. The visual shell is designed to do all of the things that the command-line shells can do.

8.2 Getting Started with the Visual Shell

This section describes how to enter, obtain help about, and leave the visual shell. This section also describes what you see on the screen while running the visual shell and how the menus work.

Note the following convention for specifying keystrokes. Ctrl refers to the Ctrl key. Ctrl-C means pressing the Ctrl and 'c' keys at the same time. ALT refers to the ALT shift key. ALT-H means pressing the ALT and 'H' keys at the same time. Note the irrelevance of case in entering Menu Selection characters. For instance, press either 'Q' or 'q' to run the "Quit" command from the main menu.

8.2.1 Entering the Visual Shell

Login to XENIX. If you are not sure how to login, consult the Operations Guide or have someone knowledgeable about XENIX help you. When you have a shell prompt (typically '$' or '%'), the operating system is waiting for a command. Enter the command:

```
vsh
```

and press RETURN.

8.2.2 Getting Help

If at anytime you are not sure what to do, either run the "Help" Menu Selection or press ALT-H. Refer to the reference section of this chapter for information about the Help command.

8.2.3 Leaving the Visual Shell

To exit the visual shell select the Quit command from the main menu. The simplest way to do this is to simply press 'q' or 'Q'. In response to the prompt "Type Y to confirm", enter 'y' or 'Y'. If you don't want to exit the visual shell yet (perhaps you pressed 'q' by mistake), enter any other character but 'y' or 'Y'. If you have invoked the visual shell from another shell, as described above, you will need to log out from XENIX by entering Ctrl-D or 'logout' and pressing RETURN. If the visual shell is your default shell, you will automatically be logged out.

8.3 The Visual Shell Screen

8.3.1 Status Line

The bottom line on the screen is called the 'status line'. The status line displays the name of the current working directory, notifies you if you have mail, and gives the date, time and the name of the operating system.

8.3.2 Message Line

The line above the 'status line' is called the 'message line'. The message line displays special output from XENIX commands, such as error reports.

8.3.3 MainMenu

The next section of the screen above the message line is the 'main menu'. The main menu displays a selection of useful XENIX commands.

The currently selected menu command is highlighted on the screen. To select any command, press the SPACE BAR. The next highlighted command is selected. The BACKSPACE key will move to the previous command. Move through the menu until you have found the command you want. To run the currently selected command, press RETURN.

You may also enter the first letter of a command to select that command. If you enter the first letter of the command, you do not need to press RETURN.

If you enter a letter which does not correspond to a menu selection, the message:

    Not a valid option

is displayed. Try another option.

8.3.4 Command Option Menu

When you have selected a command, the main menu is replaced with a command option menu. The command option menu gives the options available with the specific command. You must fill in the options with appropriate responses.

If you wish to return to the main menu without running the command, press Ctrl-C, (cancel). If you want to run the command with the selected options press RETURN.

The following keystrokes allow editing of option responses.

Ctrl-I, Ctrl-A, or TAB	Move to next field in options menu.
Ctrl-Y or DEL	Delete character under cursor.
Ctrl-L	Move cursor to character to right of current position in current option field.
Ctrl-K	Move cursor to character to left of current position in current option field.
Ctrl-P	Move cursor to word in current field to right of the current word.
Ctrl-O	Move cursor to word in current field to left of the current word.

8.3.5 Program Output

While running a command, commands given and output (unless redirected) are displayed above the menu and below the view window. The output *scrolls up:* moves from bottom to top. Lines scrolling off the top of the output window disappear.

Visual shell command lines are listed with each argument preceded by the number in the argument list enclosed in parentheses. The command is named in the output window by the menu command. Hence, if you run the command **/bin/ls** with the argument **-R**, the output window will display the command line as follows:

Run (1)/bin/ls (2) -R (3)

To change the command line format to reflect the actual XENIX command line generated by the visual shell, use the Options Output menu command.

8.3.6 View Window

A menu of currently accessible files and directories can be displayed at the top of the screen in alphabetical order, left to right, top to bottom. Note that this display is the same as that obtained using the view command. This will be referred to as the 'view window' in this chapter. If the directory list is larger than the current window size, you may scroll through using the key commands given below. To reset the window size, use the 'Window' main menu command.

The currently selected item is highlighted in the view window. Use the arrow keys and other key commands given at the end of this section to move the highlight around the window.

If a directory is being listed, subdirectories are shown enclosed in square brackets. To view a subdirectory, press '=' while the directory is highlighted. To return to the previous directory after viewing a subdirectory, press '-'. The parent directory of the current directory is shown as '[..]'. The current directory is shown as '[.]'. Executable files are preceded by an asterisk. The last modification date of the currently selected item is given at the right margin of the last line of the window. The name of the item in view in the current window is given in the upper right-hand corner of the window.

The view window may also display contents of files. Highlight a file, and press '='. You may scroll through the file using the key commands given below. While viewing a file, the highlighted area covers one line.

If you press '=' while an executable file is highlighted, that file will be run.

If the visual shell requires a file or directory name, the currently selected View Window item can be automatically entered in the relevent option field by pressing any directional movement key following selection of the command. This method saves keystrokes and reduces the chance of making a mistake while entering a command. On the other hand, if you wish to enter a file or directory in an option field, enter in the name after selecting the command.

Use these keystrokes to select files from the view window:

WINDOW MOTION KEYS

Ctrl-Q	Move to start (first item alphabetically) of view window.
Ctrl-Z	Move to end (last item alphabetically) of view window.
Ctrl-R Ctrl-E	Scroll view window up.
Ctrl-R Ctrl-S	Scroll view window down.
=	View indicated item, either file or directory. If no view window is present, the current working directory is displayed.
-	Return window display to parent directory of currently listed directory. If viewing a file, exit from viewing that file. Last view window is returned to.

DIRECTIONAL MOVEMENT KEYS

ARROWUP or Ctrl-E	Move highlight up in view window.
ARROWDOWN or Ctrl-X	Move highlight down in view window.
ARROWLEFT or Ctrl-S	Move highlight left in view window.
ARROWRIGHT or Ctrl-D	Move highlight right in view window.

Movement beyond the left or right margin will proceed to the next item on the previous or next line unless at the edge of the view window. Movement beyond the top or bottom edge of the current window will scroll the view window up or down if there are more items in that direction in the view window.

Note that there are two ways to move the highlight around. Either use the keypad arrow keys or the cluster of four keys on the far left of the keyboard 'e', 'x', 's', and 'd' shifted with Ctrl.

While viewing a file, the directional movement keys for up and left move the highlight up, and the keys for down and right move the highlighted line down.

8.4 Visual Shell Reference

8.4.1 Visual Shell Default Menu

This section describes the default visual shell menu commands and options. The menu options are displayed at the bottom of the screen above the status line.

To invoke a command, move the highlight forward through the main menu using the space bar or the tab key, or backwards using the backspace key. Or simply press the first letter of the command.

Most commands require entering options. Move the cursor to the field using the SPACE BAR, TAB key or BACKSPACE key, and enter your response. To edit the options, refer to the key commands listed above in the section in this chapter labeled "Command Option Menu". To select an item from a View Window listing for insertion in a field, refer to the section in this chapter labelled "View Window".

Note that some options have 'switches' with predefined (default) selections. The currently selected switch setting is highlighted. The default is the parenthesized setting. For instance, in the switch:

    Recursive: (yes)no

the default is recursive. To change a switch, select the field and press the SPACEBAR or BACKSPACE.

Copy

The Copy command can copy files and directories. To copy a file, select "File" from the options, to copy a directory, select "Directory". A submenu then appears. Enter the file or directory you wish copied in the *from:*

field. Enter the file or directory you wish copied to the *to:* field. Note that if the item in the *to:* field already exists, it is overwritten, so be careful.

The Copy Directory sub-menu has a switch "recursive". If this switch is set to yes, all sub-directories and their contents below the specified directory will be copied.

Delete

The Delete command can remove files and directories. In the *DELETE name:* field, enter the name of the file or directory you want to remove. Note that once the file or directory is deleted, the contents are permanently removed unless you have another copy, so be careful.

Edit

The Edit command invokes the full-screen editor **vi**. The current directory is displayed in the output window. Enter in the option field *EDIT filename:* the name of the file you wish to edit using **vi**.

To learn **vi**, refer to "vi: a Screen Editor" in the XENIX *User's Guide*, and the **vi**(C) manual page in the XENIX *Reference Manual*. A **vi** reference card is also available.

Help

The Help command (also available by pressing ALT-H at any time), can give online help regarding many aspects of visual shell use. The view window displays the help file. Use the menu to select the topic you need help with. For instance, move the highlight to 'Keyboard' using the SPACE BAR and press RETURN to view the help file starting at the 'Keyboard' section. The 'Next' and 'Previous' fields in the menu will scroll through the the help file, from the present location, one screen at a time. Your work will remain undisturbed. To return from Help, press Ctrl-C or select the 'Resume' menu option.

Mail

The Mail command enters the XENIX mail system. There are two options: "Send" and "Read" For more information about mail, refer to the section of the XENIX *Users Guide* titled "Mail", or refer to the **mail**(C) manual page.

Name

The Name command renames an existing file or directory. There are two fields, *From:* and *To:*. Enter the name of the file or directory you want to rename in *From:* and the new name in *To:*.

8.4.2 Options

The Options Main Menu Selection provides four sub-menus. These sub-menus run commands which are used infrequently, or which have irrevocable results.

Directory Option

The Directory command has two sub-menus, Make and Usage.

Make Directory Option:

This command creates a new directory named what you enter in the *name:* field.

Usage Directory Option:

Counts the number of disk blocks in the directories specified in the *name:* field. The format is the same as the XENIX command **ls -s**. Refer to the manual page **ls**(C).

FileSystem Option

FileSystem has five sub-menus: Create, FilesCheck, SpaceFree, Mount and Unmount.

Create FileSystem Option:

Create FileSystem makes a XENIX filesystem. The Create command performs radical system maintenance and may have irrevocable effects. Care is advised when using Create FileSystem.

The functionality is the same as **mkfs**(C). Consult the **mkfs**(C) manual page before running Create FileSystem. Create FileSystem prompts you for device, block size, gap number and block number. Refer to Chapter 3, "Using File Systems", in the XENIX *Operations Guide*, for information on creating file systems.

FilesCheck FileSystem Option:

FilesCheck checks the consistency of a XENIX filesystem and attempts repair if damage is detected. The FilesCheck command performs radical system maintenance and may have irrevocable effects. Care is advised when using FilesCheck.

The functionality is the same as **fsck**(C). Consult the **fsck**(C) manual page before running FilesCheck. FilesCheck prompts you for the device to check.

Output Option:

The Output Option command has one switch, *commands like: VShell XENIX* ". The default is VShell. IF VShell is set, the **vsh** form of commands given appear in the upward scrolling output window. If XENIX is specified, the XENIX command line which **vsh** generated is shown instead.

Permissions Option

The Permissions Option command allows changing the access permissions on files and directories. The functionality is the same as the **chmod**(C) command. Consult the **chmod** manual page if you do not understand the concept of XENIX permissions.

In the *name:* field enter the name of the file or directory you wish to alter the permissions on. You may only alter the permissions on files and directories you own. There are four switches, *who:*, *read:*, *write:*, and *execute:*.

The *who:* switch has four settings, *All*, *Me*, *Group* and *Others*. *All* is the default. *All* refers to yourself, those with the same group id as yourself and others. *Me* refers to yourself. *Group* refers to all others with your group id. *Others* refers to those outside your group.

The read, write and execute switches have two settings, yes and no. The default is yes for *Me*, and no for *Group* and *Others*. This grants the given type of permission to those specified in the *who:* switch. No takes away the given type of permission from those specified in the *who:* switch.

8.4.3 Print

The Print command puts a file or files in the queue for your lineprinter. In the *filename:* option field, enter the file or files you want to print.

8.4.4 Quit

The Quit command exits the visual shell. The only option is *Enter Y to confirm:*. Enter 'Y' or 'y' if you really want to quit. Any other key cancels the quit.

8.4.5 Run

The Run command executes a program or shell script. The *name:* option takes the name of an executable file. In the *parameters:* option field enter flags to pass to the executable file. The *output:* option can specify a file to redirect output to, or another program to send the output to. Enter a vertical bar '|' in the output field to use the pipe menu.

It is also possible to run an executable file by highlighting the name of the file in the View Window and pressing '='.

8.4.6 View

The View command allows you to inspect without altering the contents of files and directories. View is also available at any time for an item highlighted in the View Window by pressing '='. See the section above labelled 'View Window' for the details of using View.

To alter the height and characteristics of the View Window, use the 'Window' menu option. See the section below labelled "Window".

If you have invoked View from the menu, enter the name of the file or directory you wish to view in the *VIEW name:* field, or select from a directory view window.

To return from any View action to the previously displayed View Window, press the minus key '-'.

If you View a non-executable binary file, non-ascii characters are displayed as the character '@'.

8.4.7 Window

The Window command alters the height and redraw characteristics of the visual shell View Window.

The

WINDOWredraw: Yes (No)

switch turns redraw of the view window on or off after running a command.

The *heightinlines:* field changes the number of lines displayed in the view window. The minimum window height is 1 line. The default window height is 5 lines. The maximum window height is 15 lines.

8.4.8 Pipes

XENIX allows output from one program to be passed to another program or to be put in a file. This is called 'piping' or 'pipelining'. If the output is placed in a file it is said to be 'redirected'. Piping is supported in the visual shell through the pipe menu.

The Pipe menu is invoked by entering a vertical bar '|' character in any option field named *output:*. For instance, the Run main menu and the Pipe menu itself have an *output:* field. The available Pipe menu commands are Count, Get, Head, More, Run, Sort and Tail. Each Pipe menu sub-command also has an *output:* field, which allows construction of pipelines of arbitrary length.

8.4.9 Count

Count counts words, lines and characters in the input pipe. The default is all of the above. There is a switch for each type of item to count. The Count Pipe Menu option corresponds to the XENIX command **wc**. Consult the manual page **wc**(C) for an explanation.

8.4.10 Get

Get looks for patterns in the input pipe. The pattern may be verbatim, or you may specify a "regular expression" to look for. Regular expressions may contain 'wildcard' characters which represent sets of strings. Consult the manual page **grep** (C), for the available wildcard characters.

The first Get switch is *Unmatched (Yes) No*. If you specify Yes (the default), all lines containing the given pattern will be output. If Unmatched is set to off, all lines not containing the given pattern are output.

The second Get switch is *ignore case:* which suppresses the case while looking for the regular expression. The default is off.

The third Get switch is *line numbers:,* which reports the line in the input stream which the regular expression was matched on. The default is on.

8.4.11 Head

Head prints a specified number of lines of the input stream starting from the first line. The *lines:* field maybe set to specify the number of lines at the head of the input stream to print. The default is 5 lines.

The Head Pipe Menu option corresponds to the XENIX command head. Consult the manual page **head**(C) for an explanation.

8.4.12 More

More allows viewing an input stream one screen at a time. The More Pipe Menu option invokes the XENIX command more. Consult the manual page **more**(C) for an explanation.

8.4.13 Run

The Run Pipe Menu option allows the specification of any command not in the Pipe menu. The functionality is the same as the visual shell Main Menu Option "Run".

8.4.14 Sort

The XENIX sort utility can be invoked through the Sort Pipe menu option. The input stream is sorted.

The first Sort switch is *order:* < >. Select '>', the default, to sort in ascending order. Select '<' to sort in descending order.

The second Sort switch suppresses the case of characters in the sort. The default is off.

The third Sort switch sorts the input stream assuming an initial numeric field is in the input stream. If this switch is off, initial numbers are sorted in ascii order, which means that a line beginning with '10' will be output before the line beginning with '2'. The default is off.

The fourth Sort switch sorts the input stream in alphabetical order, rather than ascii order.

The Sort Pipe Menu option corresponds to the XENIX command sort. Consult the manual page **sort**(C) for an explanation.

8.4.15 Tail

Tail prints a specified number of lines of the input stream up to the end of the stream. The *lines:* field may be set to specify the number of lines to print. The default is 15 lines.

The Tail Pipe Menu option corresponds to the XENIX command tail. Consult the manual page **tail**(C) for an explanation.

Replace this Page
with Tab Marked:

ed

Appendix A

ed

A.1 Introduction

ed is a text editor used to create and modify text. The text is normally a document, a program, or data for a program, thus **ed** is a truly general purpose program. Note that the line editor **ex**, available with other XENIX packages is very similar to **ed**, and therefore this chapter can be used as an introduction to **ex** as well as to **ed.**

A.2 Demonstration

This section leads you through a simple session with **ed**, giving you a feel for how it is used and how it works. To begin the demonstration, invoke **ed** by entering:

```
ed
```

This invokes the editor and begins your editing session. **ed** has no prompt unless -o *string* is used on the command *line* to specify one. A blank line prompts you for commands to be entered. Initially, you are editing a temporary file that you can later copy to any file that you name. This temporary file is called the "editing buffer," because it acts as a buffer between the text you enter and the file that you will eventually write out your changes to. Typically, the first thing you will want to do with an empty buffer is add text to it. For example, after the prompt, enter:

```
a
this is line 1
this is line 2
this is line 3
this is line 4
Ctrl-D
```

This "appends" four lines of text to the buffer. To view these lines on your screen, enter:

```
1,4p
```

where the "1,4" specifies a line number range and the **p** command "prints" the specified lines on the screen.

Now enter:

```
2p
```

to view line number two. Next enter:

```
p
```

This prints out the current line on the screen, which happens to be line number two. By default, most **ed** commands operate on only the current line.

A.3 Basic Concepts

This section illustrates some of the basic concepts that you need to understand to effectively use **ed.**

A.3.1 The Editing Buffer

Each time you invoke **ed,** an area in the memory of the computer is allocated for you to perform all of your editing operations. This area is called the "editing buffer." When you edit a file, the file is copied into this buffer where you will work on the copy of the original file. Only when you write out your file, do you affect the original copy of the file.

A.3.2 Commands

Commands are entered at your keyboard. Like normal XENIX commands, entry of a command is ended by entering a NEWLINE. After you enter NEWLINE the command is carried out. In the following examples, we will presume that entry of each command is completed by entering a NEWLINE, although this will not be shown in our examples. Most commands are single characters that can be preceded by the specification of a line number or a line number range. By default, most commands operate on the "current line" described below in the section "Line Numbers." Many commands take filename or string arguments that are used by the command when it is executed.

A.3.3 Line Numbers

Any time you execute a command that changes the number of lines in the editing buffer, **ed** immediately 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 actual lines in the editing buffer that are to be affected by the given command. By default, a special line number called "dot" specifies the current line.

A.4 Tasks

This section discusses the tasks you perform in everyday editing. Frequently used and essential tasks are discussed near the beginning of this section. Seldom used and special-purpose commands are discussed later.

A.4.1 Entering and Exiting The Editor

The simplest way to invoke **ed** is to enter:

```
ed
```

The most common way, however, is to enter:

ed *filename*

where *filename* is the name of a new or existing file.

To exit the editor, all you need to do is enter:

```
q
```

If you have not yet written out the changes you have made to your file, **ed** warns you that you will lose these changes by displaying the message:

```
?
```

If you still want to quit, enter another **q**. In most cases you will want to exit by entering:

```
w
q
```

so that you first write out your changes and only *then* exit the editor.

A.4.2 Appending Text: a

Suppose that you want to create some text starting from scratch. This section shows you how to enter text in a file, just to get started. Later we'll talk about how to change it.

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

In ed terminology, the text being worked on is said to be "kept in a buffer." Think of the buffer as a workspace, or simply as a place where the information that you are going to be editing is kept. In effect, the buffer is the piece of paper on which you will write, make changes, and save (write to the disk).

You tell **ed** what to do to your text by entering instructions called "commands." Most commands consist of a single letter, each entered on a separate line. **ed** prompts with an asterisk (*).
The prompt can be turned on and off with the prompt command, **P**.

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

To enter lines of text into the buffer, enter an "a" followed by a RETURN, followed by the lines of text you want, as shown below:

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

To stop appending, enter a line that contains only a period. The period (.) tells **ed** that you have finished appending. (You can also use Ctrl-D, but we will use the period throughout this discussion.) If **ed** seems to be ignoring you, enter an extra line with just a period (.) on it. You may find you've added some garbage lines to your text, which you will have to take out later.

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, enter another **a** command, and continue entering your text.

If you make an error in the commands you enter to **ed**, it will tell you by displaying the message:

?
error message

A.4.3 Writing Out a File: w

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 that 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 text in a file named *text*, enter:

```
w text
```

Leave a space between **w** and the filename. **ed** responds by displaying the number of characters it has written out. For instance, **ed** might respond with

```
68
```

(Remember that blanks and the newline character at the end of each line are included in the character count.) Writing out a file just makes a copy of the text– the buffer's contents are not disturbed, so you can go on adding text to it. If you invoked **ed** with the command "ed *filename*," then by default, a **w** command by itself will write the buffer out to *filename*.

Note that **ed** at all times works on a copy of a file, not the file itself. No change in the contents of a file takes place until you give a **w** command. Writing out the text to a file from time to time as it is being created is a good idea. If the system crashes, or you make a mistake (not saving the file on disk), you will lose all of the text in the buffer, but any text that was written out to a file is relatively safe.

A.4.4 Leaving The Editor: q

To terminate a session with **ed**, save the text you're working on by writing it to a file using the **w** command, then enter:

```
q
```

The system responds with the XENIX prompt character. If you try to quit without writing out the file **ed** will display:

```
?
```

At that point, write out the text if you want to save it; if not, entering another "q" will get you out of the editor.

Exercise

Enter **ed** and create some text by entering:

```
a
... text ...
.
```

Write it out by entering:

w *filename*

Then leave **ed** by entering:

```
q
```

Next, use the **cat** command to display the file on your terminal screen to see that everything has worked.

A.4.5 Editing A New File: e

A common way to get text into your editing buffer is to read it in from a file. This is what you do to edit text that you have saved with the **w** command in a previous session. The **edit** (e) command places the entire contents of a file in the buffer. If you had saved the three lines "Now is the time" etc., with a **w** command in an earlier session, the **ed** command:

```
e text
```

would place the entire contents of the file *text* into the buffer and respond with

```
68
```

which is the number of characters in *text*. *If anything is already in the buffer, it is deleted first.*

If you use the **e** command to read a file into the buffer, then you don't need to use a filename after a **w** command. **ed** remembers the last filename used in an **e** command, and **w** will write to this file. Thus, a good way to operate is this:

ed
e file
[*editing session*]
w
q

This way, you can enter **w** from time to time and be secure in the knowledge that if you entered the filename right in the beginning, you are writing out to the proper file each time.

A.4.6 Changing the File to Write Out to: f

You can find out the last file written to at any time using the **file** (f) command. Just enter **f** without a filename. You can also change the name of the remembered filename with **f.** Thus, a useful sequence is:

```
ed precious
f junk
```

which gets a copy of the file named *precious*, then uses **f** to save the text in the file *junk*. The original file will be preserved as *precious*.

A.4.7 Reading in a File: r

Sometimes you want to read a file into the buffer without destroying what is already there. This function is useful for combining files. This is done with the **read** (r) command. The command:

```
r text
```

reads the file *text* into your editing buffer and adds it to the end of whatever is already in the buffer.
For example, suppose you have performed a read after an edit:

```
e text
r text
```

The buffer now contains *two* copies of *text* (i.e., six lines):

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

Like the **w** and **e** commands, after the reading operation is complete **r** prints the number of characters read in.

Exercise

Experiment with the **e** command by reading and printing various files. You may get the following error message:

> ?*name*
> cannot open input file

where *name* is the name of a nonexistent file. This means that the file doesn't exist, typically because you spelled the filename wrong, or perhaps because you do not have permission to read from or write to that file. Try alternately reading and appending, to see how they work. Verify that the command:

```
ed file.text
```

is equivalent to

```
ed
e file.text
```

A.4.8 Displaying Lines On The Screen: p

Use the "**print**"(command to print the contents of the editing buffer (or parts of it) on the terminal screen. Specify the lines where you want printing to begin and where you want it to end, separated by a comma and followed by the letter "p". Thus, to print the first two lines of the buffer (that is, lines 1 through 2) enter:

```
1,2p
```

ed displays:

```
Now is the time
for all good men
```

Suppose you want to print *all* the lines in the buffer. You could use "1,3p" as shown above if you knew there were exactly 3 lines in the buffer. But you will rarely know how many lines there are, so **ed** provides a shorthand symbol for the line number of the last line in the buffer– the dollar sign ($). Use it as shown below:

```
1,$p
```

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

```
?
interrupt
```

and waits for the next command.

To print the *last* line of the buffer, enter:

```
$p
```

You can print any single line by entering the line number, followed by a **p**. Thus:

```
1p
```

produces the response:

```
Now is the time
```

which is the first line of the buffer.

In fact, **ed** lets you abbreviate even further: you can print any single line by entering *just* the line number; there's no need to enter the letter **p**. If you enter:

```
$
```

ed prints the last line of the buffer.

You can also use **$** in combinations like:

```
$-1,$p
```

which prints the last two lines of the buffer. This helps when you want to see how far you are in your entering.

The next step is to use address arithmetic to combine the line numbers like dot (.) and dollar sign ($) with plus (+) and minus (−). (Note that "dot" is shorthand for the current line, and is discussed in a later section.) Thus:

```
$-1
```

prints the next to last line of the current file (that is, one line before the line $). For example, to recall how far you were in a previous editing session:

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

prints the last six lines. (Be sure you understand why it's six, not five.) If there aren't six lines in the file, an error message is displayed.

The command:

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

prints from three lines before the current line (line dot) to three lines after. The plus (+) can be omitted. Thus:

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

is identical in meaning.

Another area in which you can save entering effort in specifying lines is to use plus and minus as line numbers by themselves. For example:

```
-
```

by itself is a command to move back one line in the file. In fact, you can string several minus signs together to move back that many lines. For example:

```
---
```

moves back three lines, as does:

```
-3
```

Thus:

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

is also identical to

```
?.-3p+3p
```

A.4.9 Displaying The Current Line: dot (.)

Suppose your editing buffer still contains the following six lines:

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

If you enter:

```
1,3p
```

ed displays:

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

Try entering:

```
p
```

This prints:

```
to come to the aid of their party.
```

which is the third line of the buffer. In fact, it is the last (most recent) line that you have done anything with. You can repeat this **p** command without line numbers, and **ed** will continue to print line 3.

This happens because **ed** maintains a record of the last line that you did anything to (in this case, line 3, which you just printed) so that it can be used instead of an explicit line number. The line most recently acted on is referred to with a period (.) and is called "dot." Dot is a line number in the same way that dollar ($) is; it means "the current line" or loosely, "the line you most recently did something to." You can use it in several ways. One way is to enter:

```
.,$p
```

This prints all the lines from (and including) the current line clear to the end of the buffer. In our example, these are lines 3 through 6.

Some commands change the value of dot, while others do not. The **p** command sets dot to the number of the last line printed. In the example above, **p** sets dot to 6.

Dot is often used in combinations like this one:

```
.+1
```

Or equivalently:

```
.+1p
```

This means, "print the next line" and is one way of stepping slowly through the editing buffer. You can also enter:

```
.-1
```

This means, "print the line *before* the current line." This enables you to go backwards through the file if you wish. Another useful command is shown below:

```
.-3,.-1p
```

which prints the previous three lines.

Don't forget that all of these change the value of dot. You can find out what dot is at any time by entering:

```
.=
```

ed responds by printing the value of dot. Essentially, **p** can be preceded by zero, one, or two line numbers. If no line number is given, **ed** prints the "current line" the line that dot refers to. If one line number is given (with or without the letter **p**), **ed** prints that line (and dot is set there); and if two line numbers are given, **ed** prints all the lines in that range (and sets dot to the last line printed). If two line numbers are specified, the first cannot be bigger than the second.

Pressing RETURN once causes printing of the next line. It is equivalent to:

```
.+1p
```

Try it. Next, try entering a minus sign (–) by itself; it is equivalent to entering:

```
.-1p
```

Exercise

Create some text using the **a** command, and experiment with the **p** command. You will find, for example, that you can't print line 0, or a line beyond the end of the buffer, and that attempting to print lines in reverse order using "3,1p," does not work.

A.4.10 Deleting Lines: d

Suppose you want to remove three extra lines in the buffer. Use the **delete** (d) command. Its action is similar to that of **p**, except that **d** deletes lines instead of printing them. The lines to be deleted are specified for **d** exactly as they are for **p**. Thus, the command:

```
4,$d
```

deletes lines 4 through the end. There are now three lines left in our example, and you can check by entering:

```
1,$p
```

Notice that **$** now is line 3! Dot is set to the next line after the last line deleted, unless the last line deleted is the last line in the buffer. In that case, dot is set to **$**.

Exercise

Experiment with the **a**, **e**, **r**, **w**, **p**, and **d** commands until you are sure that you know what they do, and until you understand how dot (.), dollar ($), and line numbers are used.

Try using line numbers with **a**, **r**, and **w**, as well. You will find that **a** appends lines *after* the line number that you specify (rather than after dot); that **r** reads in a file *after* the line number you specify (not necessarily at the end of the buffer); and that **w** writes out exactly the lines you specify, not the whole buffer. These variations are sometimes useful. For instance, you can insert a file at the beginning of a buffer by entering:

0r *filename*

and you can enter lines at the beginning of the buffer by entering:

0a
[*input text here*]
.

Notice that entering:

```
.w
```

is very different from entering:

```
.
w
```

since the former writes out only a single line and the latter writes out the whole file.

A.4.11 Performing Text Substitutions: s

One of the most important **ed** commands is the **substitute** (s) command. This is the command that is used to change individual words or letters within a line or group of lines. It is the command used to correct spelling mistakes and entering errors.

Suppose that, due to a typing error, line 1 is:

```
Now is th time
```

The letter "e" has been left off of the word "the" You can use s to fix this up as follows:

```
1s/th/the/
```

This substitutes for the characters "th" the characters "the" in line 1. To verify that the substitution has worked, enter:

```
p
```

to get:

```
Now is the time
```

which is what you wanted. Notice that dot must be the line where the substitution took place, since the **p** command printed that line. Dot is always set this way with the s command.

The syntax for the substitute command follows:

[*starting-line,ending-line*] s/ *pattern*/ *replacement*/ *cmds*

Whatever string of characters is between the first pair of slashes is replaced by whatever is between the second pair, in *all* the lines between *starting-line* and *ending-line*. Only the first occurrence on each line is changed, however. Changing *every* occurrence is discussed later in this section. The rules for line numbers are the same as those for **p**, except that dot is set to the last line changed. (If no substitution takes place, dot is *not* changed. This displays the error message:

?
search string not found

Thus, you can enter:

1,$s/speling/spelling/

and correct the first spelling mistake on each line in the text.

If no line numbers are given, the **s** command assumes we mean "make the substitution on line dot" so it changes things only on the current line. This leads to the following sequence:

s/something/somethingelse/p

which makes a correction on the current line, then prints it to make sure the correction worked out right. If it didn't, you can try again. (Notice that the **p** is on the same line as the s command. With few exceptions, **p** can follow any command; no other multicommand lines are legal.)

It is also legal to enter:

s/string//

which means "change the first string of characters to nothing" or, in other words, remove them. This is useful for deleting extra words in a line or removing extra letters from words. For instance, if you had

Nowxx is the time

you could enter:

s/xx//p

to show:

Now is the time

Notice that two adjacent slashes mean "no characters" not a space. There *is* a difference.

Exercise

Experiment with the substitute command. See what happens if you substitute a word on a line with several occurrences of that word. For example, enter:

```
a
the other side of the coin
.
s/the/on the/p
```

This results in:

```
on the other side of the coin
```

A substitute command changes only the *first* occurrence of the first string. You can change all occurrences by adding a **g** (for "global" to the s command, as shown below:

```
s/ ... / ... /g
```

Try using characters other than slashes to delimit the two sets of characters in the s command. Anything should work except spaces or tabs.

A.4.12 Searching

Now that you have been shown the substitute command, you can move on to another important concept: context searching.

Suppose you have the original three-line text in the buffer:

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

Suppose you want to find the line that contains the word "their" so that you can change it to the word "the" With only three lines in the buffer, it's pretty easy to keep track of which line the word "their" is on. But if the buffer contains several hundred lines, and you have been making changes, deleting and rearranging lines, you would no longer really know what this line number would be. Context searching is simply a method of specifying the desired line, regardless of its number, by specifying a textual pattern contained in the line.

The way to "search for a line that contains this particular string of characters" is to enter:

```
/string of characters we want to find/
```

For example, the **ed** command:

```
/their/
```

is a context search sufficient to find the desired line. It will locate the next occurrence of the characters between the slashes (that is, "their"). Note that you do not need to enter the final slash. The above search command is the same as entering:

```
/their
```

The search command sets dot to the line on which the pattern is found and prints it for verification:

```
to come to the aid of their party.
```

"Next occurrence" means that **ed** starts looking for the string at line ".+1," searches to the end of the buffer, then continues at line 1 and searches to line dot. (That is, the search "wraps around" from **$** to 1.) It scans all the lines in the buffer until it either finds the desired line, or gets back to dot. If the given string of characters can't be found in any line, **ed** displays the error message:

```
?
search string not found
```

Otherwise, **ed** displays the line it found. You can also search *backwards* in a file for search strings by using question marks instead of slashes. For example:

```
?thing?
```

searches backwards in the file for the word "thing" as does:

```
?thing
```

This is especially handy when you realize that the string you want is backwards from the current line.

The slash and question mark are the only characters you can use to delimit a context search, though you can use any character in a substitute command. If you get unexpected results using any of the characters:

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

read Section A.5, "Context and Regular Expressions."

You can do both the search for the desired line *and* a substitution at the same time, as shown below:

```
/their/s/their/the/p
```

This displays:

```
to come to the aid of the party.
```

The above command contains three separate actions. The first is a context search for the desired line, the second is the substitution, and the third is the printing of the line.

The expression "/their/" is a context search expression. In their simplest form, all context search expressions are a string of characters surrounded by slashes. Context searches are interchangeable with line numbers, so they can be used by themselves to find and print a desired line, or as line numbers for some other command, like s. They were used both ways in the previous examples.

Suppose the buffer contains the three familiar lines:

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

The **ed** line numbers:

```
/Now/+1
/good/
/party/-1
```

are all context search expressions, and they all refer to the same line (line 2). To make a change in line 2, enter:

```
/Now/+1s/good/bad/
```

or

```
/good/s/good/bad/
```

or

```
/party/-1s/good/bad/
```

The choice is dictated only by convenience. For instance, you could print all three lines by entering:

```
/Now/,/party/p
```

or

```
/Now/,/Now/+2p
```

or any similar combination. The first combination is better if you don't know how many lines are involved.

The basic rule is that a context search expression is the same as a line number, so it can be used wherever a line number is needed.

Suppose you search for:

```
/listing/
```

and when the line is printed, you discover that it isn't the "listing" that you wanted, so it is necessary to repeat the search. You don't have to reenter the search, because the construction:

```
//
```

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

```
??
```

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

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

```
/listing/
```

ed prints the line containing "listing".

```
s//good/p
```

This changes "listing" to "good." To go backwards and change "listing" to "good" enter:

```
??s//good/
```

Exercise

Experiment with context searching. Scan through a body of text with several occurrences of the same string of characters using the same context search.

Try using context searches as line numbers for the substitute, print, and delete commands. (Context searches can also be used with the **r**, **w**, and **a** commands.)

Try context searching using *?text?* instead of */ text /*. This scans lines in the buffer in reverse order instead of normal order, which is sometimes useful if you go too far while looking for a string of characters. It's an easy way to back up in the file you're editing.

If you get unexpected results with any of the characters

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

read Section A.5, "Context and Regular Expressions."

A.4.13 Changing and Inserting Text: c and i

This section discusses the **change** (c) command, which is used to change or replace one or more lines, and the **insert** (i) command, which is used for inserting one or more lines.

The **c** command is used to replace a number of lines with different lines that you type at the terminal. For example, to change lines ".+1" through "$" to something else, enter:

.+1,$c
type the lines of text you want here...
.

The lines you enter between the **c** command and the dot (.) will replace the originally addressed lines. This is useful in replacing a line or several lines that have errors in them.

If only one line is specified in the **c** command, then only that line is replaced. (You can enter as many replacement lines as you like.) Notice the use of a period to end the input. This works just like the period in the append command and must appear by itself on a new line. If no line number is given, the current line specified by dot is replaced. The value of dot is set to the last line you typed in. Note that the terminating period and the line referenced by dot are completely different: the first is used simply to terminate a command, the second points at a specific line of text.

The **i** command is similar to the append command. For example:

```
/string/i
type the lines to be inserted here ...
```

inserts the given text *before* the next line that contains "string." The text between **i** and the terminating period is *inserted before* the specified line. If no line number is specified, dot is used. Dot is set to the last line inserted.

Exercise

The **c** command is like a combination of delete followed by insert. Experiment to verify that:

start,end d
i
[*text*]
.

is almost the same as:

start,end c
[*text*]
.

These are not *precisely* the same, if the last line gets deleted.

Experiment with **a** and **i** to see that they are similar, but not the same. Observe that:

line-number a
[*text*]
.

appends *after* the given line, while:

line-number i
[*text*]
.

inserts *before* it. If no line number is given, **i** inserts before line dot, while **a** appends after line dot.

A.4.14 Moving Lines: m

The **move** (m) command lets you move a group of lines from one place to another in the buffer. Suppose you want to put the first three lines of the buffer at the end instead. You *could* do it by entering:

```
1,3w temp
$rtemp
1,3d
```

where *temp* is the name of a temporary file. However, you can do it easily with the **m** command:

```
1,3m$
```

This will move lines 1 through 3 to the end of the file.

The general case is:

*start-line,end-line*m*after-this-line*

There is a third line to be specified: the place where the moved text gets put. Of course, the lines to be moved can be specified by context searches. If you had:

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

you could reverse the two paragraphs like this:

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

Notice the −1. The moved text goes *after* the line mentioned. Dot gets set to the last line moved. Your file will now look like this:

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

As another example of a frequent operation, you can reverse the order of two adjacent lines by moving the first line after the second line. Suppose that you are positioned at the first line. Then:

```
m+
```

moves line dot to one line after the current line dot. If you are positioned on the second line:

```
m--
```

moves line dot to one line after the current line dot.

The **m** command is more efficient 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. The result of a bad **m** command can be a mess. Doing the job one step at a time makes it easier for you to verify, at each step, that you accomplished what you wanted. It is also a good idea to issue a **w** command before doing anything complicated; then if you make a mistake, it's easy to back up to where you were.

For more information on moving text, see Section A.4.18, "Marking Your Spot in a File:k" in this Appendix.

A.4.15 Performing Global Commands: g and v

The "global" commands **g** and **v** are used to execute one or more editing commands on all lines that either contain **g** or do not contain **v**, a specified pattern.

For example, the command:

```
g/XENIX/p
```

prints all lines that contain the word "XENIX." The pattern that goes between the slashes can be anything that could be used in a line search or in a substitute command; exactly the same rules and limitations apply.

For example:

```
g/^\./p
```

prints all the **troff** formatting commands in a file. "..". For an explanation of the use of the caret (^) and the backslash (\), see Section A.5, "Context and Regular Expressions" in this Appendix.

The **v**, command is identical to **g**, except that it operates on those lines that do *not* contain an occurrence of the pattern. (Mnemonically, the "v" can be thought of as part of the word "in **v** erse".

For example:

```
v/^\./p
```

prints all the lines that do not begin with a period (i.e., the actual text lines).

Any command can follow **g** or **v**. For example, the following command deletes all lines that begin with ".."

```
g/^\./d
```

This command deletes all empty lines:

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

Probably the most useful command that can follow a global command is the substitute command. For example, we could change the word "Xenix" to "XENIX" everywhere, and verify that it really worked, with:

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

Notice that we used // in the substitute command.to mean "the previous pattern" in this case, "Xenix." The **p** command executes on each line that matches the pattern, not just on those in which a substitution took place.

The global command makes two passes over the file. On the first pass, all lines that match the pattern are marked. On the second pass, each marked line is examined in turn, dot is set to that line, and the command executed. This means that it is possible for the command that follows a **g** or **v** command to use addresses, set dot, and so on, quite freely. For example:

```
g/^\.P/+
```

prints the line that follows each ".P" command (the signal for a new paragraph in some formatting packages). Remember that plus (+) means "one line past dot." And:

```
g/topic/?^\.H?p
```

searches for each line that contains the word "topic" scans backwards until it finds a line that begins with a ".H" (a heading) and prints it, thus showing the headings under which "topic" is mentioned. Finally:

```
g/^\.EQ/+,/^\.EN/-p
```

prints all the lines that lie between lines beginning with ".EQ" and ".EN" formatting commands.

The **g** and **v** commands can also be preceded by line numbers, in which case the lines searched are only those in the range specified.

It is possible to give more than one command under the control of a global command. For example, suppose the task is to change "x" to "y" and "a" to "b" on all lines that contain "thing." Then:

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

is sufficient. The backslash (\) signals the **g** command that the set of commands continues on the next line; the **g** command terminates on the first line that does not end with a backslash.

Note that you cannot use a substitute command to insert a newline within a **g** command. Watch out for this.

The command:

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

does *not* work as you might expect. The remembered pattern is the last pattern that was actually executed, so sometimes it will be "x" (as expected), and sometimes it will be "a" (not expected). You must spell it out, as shown:

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

It is also possible to execute **a**, **c** and **i** commands as part of a global command. As with other multiline constructions, add a backslash at the end of each line except the last. Thus, to add an ".nf" and ".sp" command before each ".EQ" line, enter:

```
g/^\.EQ/i\
.nf\
.sp
```

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

A.4.16 Displaying Tabs and Control Characters: l

ed provides two commands for printing the contents of the text you are editing. You should already be familiar with **p**, in combinations like:

```
1,$p
```

to print all the lines you are editing, or:

```
s/abc/def/p
```

to change "abc" to "def" on the current line. Less familiar is the "list" (**l**) command which gives slightly more information than **p**. In particular, **l** makes visible characters that are normally invisible, such as tabs and backspaces. If you list a line that contains some of these, **l** prints each tab as ">" and each backspace as "<" This makes it much easier to correct the sort of entering mistake that inserts extra spaces adjacent to tabs, or inserts a backspace followed by a space.

The **l** command also "folds" long lines for printing. Any line that exceeds 72 characters is printed on multiple lines; each printed line except the last is terminated by a backslash (\), so you can tell it was folded. This is useful for printing lines longer than the width of your terminal screen.

Occasionally, the **l** command will print a string of numbers preceded by a backslash, such as \07 or \16. These combinations are used to make visible characters that normally don't print, like form feed, vertical tab, or bell. Each backslash-number combination represents a single ASCII character. Note that numbers are octal and not decimal. When you see such characters, be aware that they may have surprising meanings when printed on some terminals. Often, their presence indicates an error in entering, because they are rarely used.

A.4.17 Undoing Commands: u

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

```
u
```

This command does *not* work with the g and v commands.

A.4.18 Marking Your Spot in a File: k

The mark command, **k**, provides a facility for marking a line with a particular name, so that you can later reference it by name, regardless of its actual line number. This can be handy for moving lines and keeping track of them as they move. For example:

```
kx
```

marks the current line with the name "x." If a line number precedes the **k**, that line is marked. (The mark name must be a single lowercase letter.) You can refer to the marked line with the notation:

```
´x
```

Note the use of the single quotation mark (´) here. Marks are very useful for moving things around. Find the first line of the block to be moved and then mark it with:

```
ka
```

Then find the last line and mark it with:

```
kb
```

Go to the place where the text is to be inserted and enter:

```
´a,´bm.
```

A line can have only one mark name associated with it at any given time.

A.4.19 Transferring Lines: t

We mentioned earlier the idea of saving lines that are hard to type or used often, to cut down on entering time. ed provides another command, called **t** (for transfer) for making a copy of a group of one or more lines at any point. This is often easier than writing and reading.

The **t** command is identical to the **m** command, except that instead of moving lines it simply duplicates them at the place you named. Thus:

```
1,$t$
```

duplicates the entire contents that you are editing.

A common use for **t** is to create a series of lines that differ only slightly. For example, you can enter:

```
a
Now is the time for all good men to come to the aid of their party.
.
t.                      [make a copy]
s/men/women/            [change it a bit]
t.                      [make third copy]
s/Now is/yesterday was/ [change it a bit]
```

Your file will look like this:

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

A.4.20 Escaping to the Shell: !

Sometimes it is convenient to temporarily escape from the editor to execute a XENIX command without leaving the editor. The shell **escape** (!) command, provides a way to do this.

If you enter:

!*command*

your current editing state is suspended, and the XENIX command you asked for is executed. When the command finishes, **ed** will signal you by printing another exclamation (!). At that point, you can resume editing.

A.5 Context and Regular Expressions

You may have noticed that things don't work right when you use characters such as the period (.), the asterisk (*), and the dollar sign ($) in context searches and with the substitute command. The reason is rather complex, although the solution to the problem is simple. ed treats these characters as special. For instance, in a context search or the first string of the substitute command, the period (.) means "any character" not a period, so:

/x.y/

means a line with an "x" any character, and a "y" not just a line with an "x" a period, and a "y" A complete list of the special characters that can cause problems follows:

^ . $ [* \ /

The next few subsections discuss how to use these characters 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*(C), *sed*(C) (See the XENIX *Reference Manual*).

Recall that a trailing g after a substitute command causes all occurrences to be changed. With:

s/this/that/

and

s/this/that/g

The first command replaces the *first* "this" on the line with "that." If there is more than one "this" on the line, the second form with the trailing g changes *all* of them.

Either form of the **s** command can be followed by **p** or **l** to print or list the contents of the line. For example, all of the following are legal and mean slightly different things:

```
s/this/that/p
s/this/that/l
s/this/that/gp
s/this/that/gl
```

Make sure you know what the differences are.

Of course, any **s** command can be preceded by one or two line numbers to specify that the substitution is to take place on a group of lines. Thus:

```
1,$s/mispell/misspell/
```

changes the *first* occurrence of "mispell" to "misspell" in each line of the file. But:

```
1,$s/mispell/misspell/g
```

changes *every* occurrence in each line (and this is more likely to be what you wanted).

If you add a **p** or **l** to the end of any of these substitute commands, only the last line changed is printed, not all the lines. We will talk later about how to print all the lines that were modified.

A.5.1 Period: (.)

The first metacharacter that we will discuss is the period (.). On the left side of a substitute command, or in a search, a period stands for *any* single character. Thus the search:

```
/x.y/
```

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

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

and so on.

Since a period matches a single character, it gives you a way to deal with funny characters printed by **l**. Suppose you have a line that appears as:

```
th\07is
```

when printed with the **l** command, and that you want to get rid of the \07, which represents an ASCII bell character.

The most obvious solution is to enter:

```
s/\07//
```

but this will fail. Another solution is to retype the entire line. This is guaranteed, and is actually quite reasonable if the line in question isn't too big. But for a very long line, reentering is not the best solution. This is where the metacharacter ".." comes in handy. Since \07 really represents a single character, if we enter:

```
s/th.is/this/
```

the job is done. The period matches the mysterious character between the "h" and the "i" whatever it is.

Since the period matches any single character, the command:

```
s/./,/
```

converts the first character on a line into a comma (,), which very often is not what you intended. The special meaning of the period can be removed by preceding it with a backslash.

As is true of many characters in ed, the period (.) has several meanings, depending on its context. This line shows all three:

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

The first period is the line number of the line we are editing, which is called "dot." The second period is a metacharacter that matches any single character on that line. The third period is the only one that really is an honest, literal period. (Remember that a period is also used to terminate input from the **a** and **i** commands.) On the *right* side of a substitution, the period (.) is not special. If you apply this command to the line:

Now is the time.

the result is:

.ow is the time.

which is probably not what you intended. To change the period at the end of the sentence to a comma, enter:

s/\./,/

The special meaning of the period can be removed by preceding it with a backslash.

A.5.2 Backslash: \

Since a period means "any character" the question naturally arises: what do you do when you really want a period? For example, how do you convert the line:

Now is the time.

into

Now is the time?

The backslash (\), turns off any special meaning that the next character might have; in particular, "\" converts the "." from a
"match anything"
into a literal period, so you can use it to replace the period in "Now is the time." like this:

s/\./?/

The pair of characters "\." is considered by ed to be a single real period.

The backslash can also be used when searching for lines that contain a special character. Suppose you are looking for a line that contains:

at the start of a line. The search:

/.DE/

isn't adequate, for it will find lines like:

```
JADE
FADE
MADE
```

because the "." matches the letter "A" on each of the lines in question. But if you enter:

```
/\.DE/
```

only lines that contain ".DE" are found.

The backslash can be used to turn off special meanings for characters other than the period. For example, consider finding a line that contains a backslash. The search:

```
/\/
```

will not work, because the backslash (\ ) isn't a literal backslash, but instead means that the second slash (/) no longer delimits the search. By preceding a backslash with another backslash, you can search for a literal backslash:

```
/\\/
```

You can search for a forward slash (/) with:

```
/\//
```

The backslash turns off the special meaning of the slash immediately following, so that it doesn't terminate the slash-slash construction prematurely.

A miscellaneous note about backslashes and special characters: you can use any character to delimit the pieces of an s command; there is nothing sacred about slashes. (But you must use slashes for context searching.) For instance, in a line that contains several slashes already, such as:

```
//exec //sys.fort.go // etc...
```

you could use a colon as the delimiter. To delete all the slashes, enter:

```
s:/::g
```

The result is:

```
exec sys.fort.go etc...
```

When you are adding text with **a** or **i** or **c**, the backslash has no special meaning, and you should only put in one backslash for each one you want.

Exercise

Find two substitute commands, each of which converts the line:

```
\x\.\y
```

into the line:

```
\x\y
```

Here are several solutions; you should verify that each works:

```
s/\\\.//
s/x../x/
s/..y/y/
```

A.5.3 Dollar Sign: $

The dollar sign "$" stands for "the end of the line." Suppose you have the line:

```
Now is the
```

and you want to add the word "time" to the end. Use the dollar sign ($) as shown below:

```
s/$/time/
```

to get:

```
Now is the time
```

A space is needed before "time" in the substitute command, or you will get:

    Now is thetime

You can replace the second comma in the following line with a period without altering the first.

    Now is the time, for all good men,

The command needed is:

    s/,$/./

to get:

    Now is the time, for all good men.

The dollar sign ($), here, provides context to make specific which comma we mean. Without it, the s command would operate on the first comma to produce:

    Now is the time. for all good men,

To convert:

    Now is the time.

into:

    Now is the time?

as we did earlier, we can use:

    s/.$/?/

Like the period (.), the dollar sign ($) has multiple meanings depending on context. In the following line:

    $s/$/$/

the first "$" refers to the last line of the file, the second refers to the end of that line, and the third is a literal dollar sign to be added to that line.

A.5.4 Caret: ^

The caret (^) stands for the beginning of the line. For example, suppose you are looking for a line that begins with "the." If you enter:

```
/the/
```

you will probably find several lines that contain "the" in the middle before arriving at the one you want. But, by entering:

```
/^the/
```

you narrow the context, and thus arrive at the desired line more easily.

The other use of the caret (^) enables you to insert something at the beginning of a line. For example:

```
s/^/ /
```

places a space at the beginning of the current line.

Metacharacters can be combined. To search for a line that contains *only* the characters:

```
.P
```

you can use the command:

```
/^\.P$/
```

A.5.5 Star: *

Suppose you have a line that looks like this:

```
text x    y text
```

where "text" stands for lots of text, and there are an indeterminate number of spaces between the "x" and the "y." Suppose the job is to replace all the spaces between "x" and "y" with a single space. The line is too long to retype, and there are too many spaces to count.

This is where the metacharacter "star" (*) comes in handy. A character followed by a star stands for as many consecutive occurrences of that character as possible. To refer to all the spaces at once, enter:

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

The " " means "as many spaces as possible." Thus "x *y" means an "x" as many spaces as possible, then a "y"

The star can be used with any character, not just a space. If the original example was:

    text x---------ytext

then all minus signs (–) can be replaced by a single space with the command:

    s/x-*y/xy/

Finally, suppose that the line was:

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

If you enter:

    s/x.*y/xy/

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

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

then entering:

    s/x.*y/xy/

takes everything from the *first* "x" to the *last* "y" which, in this example, is more than you wanted.

The solution is to turn off the special meaning of the period (.) with the backslash (\):

    s/x\.*y/xy/

Now the substitution works, for "\.*" means "as many periods as possible."

There are times when the pattern ".*" is exactly what you want. For example, to change:

    Now is the time for all good men ....

into:

Now is the time.

use ".*" to remove everything after the "for."

s/ for.*/./

There are a couple of additional pitfalls associated with the star (*). Most notable is the fact that "as many as possible" means *zero* or more. The fact that zero is a legitimate possibility, is sometimes rather surprising. For example, if our line contained:

xy□text□x□□y□text

where the squares represent spaces, and we entered:

s/x□*y/x□y/

the first "xy" matches this pattern, for it consists of an "x" zero spaces, and a "y." The result is that the substitute acts on the first "xy" and does not touch the later one that actually contains some intervening spaces.

The way around this is to specify a pattern like:

/x□□*y/

which says an "x" a space, then as many more spaces as possible, and then a "y" (i.e., one or more spaces).

The other pitfall associated with the star (*) again relates to the fact that zero is a legitimate number of occurrences of something followed by a star. The command:

s/x*/y/g

when applied to the line:

abcdef

produces:

yaybycydyeyfy

which is almost certainly not what was intended. The reason for this is that zero is a legitimate number of matches, and there are no x's at the beginning of the line (so that gets converted into a "y," nor between the "a" and the "b" (so that gets converted into a "y," and so on. If you don't want zero matches, enter:

```
s/xx*/y/g
```

since "xx*" is one or more x's.

A.5.6 Brackets: [and]

Suppose that you want to delete any numbers that appear at the beginning of all lines of a file. You might try a series of commands like:

```
1,$s/^1*//
1,$s/^2*//
1,$s/^3*//
```

and so on, but this is clearly going to take forever if the numbers are long. Unless you want to repeat the commands over and over, until finally all the numbers are gone, you must get all the digits on one pass. That is the purpose of the brackets.

The construction:

```
[0123456789]
```

matches any single digit; the whole thing is called a "character class." With a character class, the job is easy. The pattern "[0123456789]*" matches zero or more digits (an entire number), so:

```
1,$s/^[0123456789]*//
```

deletes all digits from the beginning of all lines.

Any characters can appear within a character class, and there are only three special characters (^,], and –) inside the brackets; even the backslash doesn't have a special meaning. To search for special characters, for example, you can enter:

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

It's a nuisance to have to spell out the digits, so you can abbreviate them as [0–9]; similarly, [a–z] stands for the lowercase letters, and [A–Z] for uppercase.

Within [ ], the "[" is not special. To get a "]" (or a "–" into a character class, make it the first character.

You can also specify a class that means "none of the following characters." This is done by beginning the class with a caret (^). For example:

    [^0-9]

stands for "any character *except* a digit." Thus, you might find the first line that doesn't begin with a tab or space with a search like:

    /^[^(space)(tab)]/

Within a character class, the caret has a special meaning only if it occurs at the beginning. Verify that:

    /^[^^]/

finds a line that doesn't begin with a caret.

A.5.7 Ampersand: &

To save entering, the ampersand (&) can be used in substitutions to signify the string of text that was found on the left side of a substitute command. Suppose you have the line:

    Now is the time

and you want to make it:

    Now is the best time

You can enter:

    s/the/the best/

It's unnecessary to repeat the word "the." The ampersand (&) eliminates this repetition. On the *right* side of a substitution, the ampersand means "whatever was just matched" so you can enter:

    s/the/& best/

and the ampersand will stand for "the." This isn't much of a saving if the thing matched is just "the" but if the match is very long, or if it is something like ".*" which matches a lot of text, you can save some tedious entering. There is also much less chance of making an entering error in the replacement text. For example, to put parentheses in a line, regardless of its length, enter:

```
s/.*/(&)/
```

The ampersand can occur more than once on the right side. For example:

```
s/the/& best and & worst/
```

makes:

```
Now is the best and the worst time
```

and:

```
s/.*/&? &!!/
```

converts the original line into:

```
Now is the time? Now is the time!!
```

To get a literal ampersand, use the backslash to turn off the special meaning. For example:

```
s/ampersand/\&/
```

converts the word into the symbol. The ampersand is not special on the left side of a substitute command, only on the right side.

A.5.8 Substituting New Lines

ed provides a facility for splitting a single line into two or more shorter lines by "substituting in a newline." For example, suppose a line has become unmanageably long because of editing. If it looks like:

```
....text xy text.....
```

you can break it between the "x" and the "y" like this:

```
s/xy/x\
y/
```

This is actually a single command, although it is entered on two lines. Because the backslash (\) turns off special meanings, a backslash at the end of a line makes the newline there no longer special.

You can, in fact, make a single line into several lines with this same mechanism. As an example, consider italicizing the word "very" in a long line by splitting "very" onto a separate line, and preceding it with the formatting command ".I." Assume the line in question looks like this:

```
text a very big text
```

The command:

```
s/ very /\
.I\
very\
/
```

converts the line into four shorter lines, preceding the word "very" with the line ".I" and eliminating the spaces around the "very" at the same time.

When a newline is substituted in a string, dot is left at the last line created.

A.5.9 Joining Lines

Lines may be joined together, with the **j** command. Assume that you are given the lines:

```
Now is
 the time
```

Suppose that dot is set to the first line. Then the command:

```
j
```

joins them together to produce:

```
Now is the time
```

No blanks are added, which is why a blank was shown at the beginning of the second line.

All by itself, a **j** command joins the lines signified by dot and dot+1, but any contiguous set of lines can be joined. Just specify the starting and ending line numbers. For example:

```
1,$jp
```

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

A.5.10 Rearranging a Line: \(and \)

Recall that "&" is shorthand for whatever was matched by the left side of an s command. In much the same way, you can capture separate pieces of what was matched. The only difference is that you have to specify on the left side just what pieces you're interested in.

Suppose that you have a file of lines that consist of names in the form:

```
Smith, A. B.
Jones, C.
```

and so on, and you want the initials to precede the name, as in:

```
A. B. Smith
C. Jones
```

It is possible to do this with a series of editing commands, but it is tedious and error-prone.

The alternative is to "tag" the pieces of the pattern (in this case, the last name, and the initials), then rearrange the pieces. On the left side of a substitution, if part of the pattern is enclosed between \(and \), whatever matched that part is remembered, and available for use on the right side. On the right side, the symbol, "\1" refers to whatever matched the first \(...\) pair; "\2" to the second \(...\), and so on.

The command:

```
1,$s/^\([.*]\), *\(.*\)/\2\1/
```

although hard to read, does the job. The first \(...\), matches the last name, which is any string up to the comma; this is referred to on the right side with "\1." The second \(...\), is whatever follows the comma and any spaces, and is referred to as "\2."

With any editing sequence this complicated, it is unwise to simply run it and hope. The global commands, **g** and **v**, provide a way for you to print exactly those lines which were affected by the substitute command, and thus, verify that it did what you wanted in all cases.

A.6 Speeding Up Editing

One of the most effective ways to speed up your editing is knowing what lines will be affected by a command. If you do not specify the lines it is to act on, and on what line you will be positioned (i.e., the value of dot) when a command finishes, your editing speed is slowed. If you can edit without specifying unnecessary line numbers, you can save a lot of entering.

For example, if you issue a search command like:

```
/thing/
```

you are left pointing at the next line that contains "thing." Then no address is required with commands like **s**, to make a substitution on that line, or **p**, to print it, or **l**, to list it, or **d**, to delete it, or **a**, to append text after it, or **c**, to change it, or **i**, to insert text before it.

What happens if there is no occurrence of "thing." Dot is unchanged. This is also true if the cursor was on the only occurrence of "thing" when you issued the command. The same rules hold for searches that use ?...?; the only difference is the direction in which you search.

The delete command, **d**, leaves dot pointing at the line that followed the last deleted line. When the line dollar ($) gets deleted, however, dot points at the *new* line $.

The line-changing commands **a**, **c**, and **i**, by default, all affect the current line. If you give no line number with them, **a** appends text after the current line, **c** changes the current line, and **i** inserts text before the current line.

The **a**, **c**, and **i** commands behave identically in one respect; when you stop appending, changing or inserting, dot points at the last line entered. This is exactly what you want when entering and editing on the fly.
For example, you can enter:

```
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 enter:

```
a
text
horrible botch  (major error)
.
c       (replace entire line)
fixed up line
.
```

Experiment to determine what happens if you add *no* lines with an **a**, **c**, or **i** command.

The **r** command reads a file into the text being edited, at the end if you give no address, or after the specified line if you do. In either case, dot points at the last line read in. Remember that you can even enter:

```
0r
```

to read a file in at the beginning of the text. (You can also enter *0a* or *1i* to start adding text at the beginning.)

The **w** command writes out the entire file. If you precede the command by one line number, that line is written out. If you precede it by two line numbers, that range of lines is written out. The **w** command does *not* change dot: the current line remains the same, regardless of what lines are written out. This is true even if you enter something like:

```
/^\.AB/,/^\.AE/w abstract
```

which involves a context search.

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

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

A.6.1 Semicolon: ;

Searches with /.../ and ?...? start at the current line and move forward or backward, respectively, until they either find the pattern, or get back to the current line. Sometimes, this is not what you want. Suppose, for example, that the buffer contains lines like this:

```
.
.
.
ab
.
.
.
bc
.
.
.
```

Starting at line 1, you would expect the command:

```
/a/,/b/p
```

to print all the lines from the "ab" to the "bc" inclusive. This is not what happens. *Both* searches (for "a" and for "b" start from the same point, and thus, they both find the line that contains "ab." As a result, a single line is printed. Worse, if there had been a line with a "b" in it before the "ab" line, then the print command would be in error, since the second line number would be less than the first, and it is illegal to try to print lines in reverse order.

This is because the comma separator for line numbers doesn't set dot as each address is processed; each search starts from the same place. In ed, the semicolon (;) can be used just like the comma, with the single difference that use of a semicolon forces dot to be set at the time the semicolon is encountered, as the line numbers are being evaluated. In effect, the semicolon "moves" dot. Thus, in our example above, the command:

```
/a/;/b/p
```

prints the range of lines from "ab" to "bc" because after the "a" is found, dot is set to that line, and then "b" is searched for, starting beyond that line.

This property is most useful in a very simple situation. Suppose you want to find the *second* occurrence of "thing." You could enter:

```
/thing/
//
```

but this prints the first occurrence as well as the second, and is a nuisance when you know very well that it is only the second one you're interested in. The solution is to enter:

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

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

Closely related is searching for the second to last occurrence of something, as in:

```
?something?;??
```

Finally, bear in mind that if you want to find the first occurrence of something in a file, starting at an arbitrary place within the file, it is not sufficient to enter:

```
1;/thing/
```

because, if "thing" occurs on line 1, it will not be found. The command:

```
0;/thing/
```

will work because it starts the search at line 1. This is one of the few places where 0 is a legal line number.

A.6.2 Interrupting the editor

As a final note on what dot gets set to, you should be aware that if you press the INTERRUPT key while **ed** is executing a command, your file is restored, as much as possible, to what it was before the command began. Naturally, some changes are irrevocable; if you are reading in or writing out a file, making substitutions, or deleting lines. These will be stopped in some unpredictable state in the middle (which is why it usually is unwise to stop them). Dot may or may not be changed.

If you are using the print command, dot is not changed until the printing is done. Thus, if you decide to print until you see an interesting line, and then press INTERRUPT, to stop the command, dot will *not* be set to that line or even near it. Dot is left where it was when the **p** command was started.

A.7 Cutting and Pasting with the editor

This section describes how to manipulate pieces of files, individual lines or groups of lines.

A.7.1 Inserting One File Into Another

Suppose you have a file called *memo*, and you want the file called *table* to be inserted just after a reference to Table 1. That is, in *memo*, somewhere is a line that reads:

Table 1 shows that ...

and the data contained in *table* has to go there.

To put *table* into the correct place in the file edit *memo*, find "Table 1" and add the file *table* right there:

```
ed memo
/Table 1/
response from ed
.r table
```

The critical line is the last one. The **r** command reads a file; here you asked for it to be read in right after line dot. An **r** command, without any address, adds lines at the end, so it is the same as "$r."

A.7.2 Writing Out Part of a File

The other side of the coin is writing out part of the document you're editing. For example, you may want to split the table from the previous example into a separate file so it can be formatted and tested separately. Suppose that in the file being edited we have:

```
.TS
[lots of stuff]
.TE
```

which is the way a table is set up for the **tbl** program. To isolate the table in a separate file called *table*, first find the start of the table (the ".TS" line), then write out the interesting part. For example, first enter:

```
/^\.TS/
```

This prints out the found line:

```
.TS
```

Next enter:

```
.,/^\.TE/wtable
```

and the job is done. Note that you can do it all at once with:

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

The point is that the w command can write out a group of lines, instead of the whole file. In fact, you can write out a single line if you like; just give one line number instead of two. If you have just entered a complicated line and you know that it (or something like it) is going to be needed later, then save it, do not retype it.
For example, in the editor, enter:

```
a
lots of stuff
horrible line
.
.w temp
a
more stuff
.
.r temp
a
more stuff
.
```

A.8 Editing Scripts

If a fairly complicated set of editing operations is to be done on a whole set of files, the easiest thing to do is to make up a "script" (i.e., a file that contains the operations you want to perform, then apply this script to each file in turn).

For example, suppose you want to change every "Xenix" to "XENIX" and every "USA" to "America" in a large number of files. Enter the following lines into the file *script:*

```
g/Xenix/s//XENIX/g
g/USA/s//America/g
w
q
```

Now you can enter:

```
ed - file1 <script
ed - file2 <script
...
```

This causes **ed** to take its commands from the prepared file *script.* Notice that the whole job has to be planned in advance, and that by using the XENIX shell command interpreter, you can cycle through a set of files automatically. The dash (–) suppresses unwanted messages from **ed.**

When preparing editing scripts, you may need to place a period as the only character on a line to indicate termination of input from an **a** or **i** command. This is difficult to do in ed, because the period you type will terminate input rather than be inserted in the file. Using a backslash to escape the period won't work either. One solution is to create the script using a character such as the at-sign (@), to indicate end of input. Then, later, use the following command to replace the at-sign with a period:

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

A.9 Summary of Commands

This following is a list of all **ed** commands. The general form of **ed** commands is the command name, preceded by one or two optional line numbers and, in the case of **e**, **f**, **r**, and **w**, followed by a filename. Only one command is allowed per line, but a **p** command may follow any other command (except **e**, **f**, **r**, **w**, and **q**).

a Appends, i.e., adds lines to the buffer (at line dot, unless a different line is specified). Appending continues until a period is entered on a new line. The value of dot is set to the last line appended.

c Changes the specified lines to the new text which follows. The new lines are terminated by a period on a new line, as with **a**. If no lines are specified, replace line dot. Dot is set to the last line changed.

d Deletes the lines specified. If none are specified, deletes line dot. Dot is set to the first undeleted line following the deleted lines unless dollar ($) is deleted, in which case dot is set to dollar.

e Edits a new file. Any previous contents of the buffer are thrown away, so issue a **w** command first.

f Prints the remembered filename. If a name follows **f**, then the remembered name is set to it.

g The command *g/ string /commands* executes *commands* on those lines that contain *string*, which can be any context search expression.

i Inserts lines before specified line (or dot) until a single period is typed on a new line. Dot is set to the last line inserted.

l Lists lines, making visible nonprinting ASCII characters and tabs. Otherwise similar to **p**.

m Moves lines specified to after the line named after m. Dot is set to the last line moved.

p Prints specified lines. If none are specified, print the line specified by dot. A single line number is equivalent to a 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 it twice in a row to abort edit.

r Reads a file into buffer (at end unless specified elsewhere). Dot is set to the last line read.

s The command "s/ *string1* / *string2* /" substitutes the pattern matched by *string1* with the string specified by *string2* in the specified lines. If no lines are specified, the substitution takes place only on the line specified by dot. Dot is set to the last line in which a substitution took place, which means that if no substitution takes place, dot remains unchanged. The s command changes only the first occurrence of *string1* on a line; to change multiple occurrences on a line, enter a **g** after the final slash.

t Transfers specified lines to the line named after **t**. Dot is set to the last line moved.

v The command *v/ string / commands executes commands on those lines that donot contain string.*

u Undoes the last substitute command.

w Writes out theeditingbufferto a file. Dot remains unchanged.

.= Prints value of dot. (An equal sign by itself prints the value of $.)

!command
The line *!cmd-line* causes *cmd-line* to be executed as a XENIX command.

/ string /
Context search. Searches for next line which contains this string of characters and prints it. Dot is set to the line where string was found. The search starts at .+1, wraps around from $ to 1, and continues to dot, if necessary.

? string ?
Context search in reverse direction. Starts search at .–1 , scans to 1, wraps around to $.

Index

CHARACTERS

A

B

D

E

F

G

H

I

N

O

P

Q

R

S

V

03-17-87

SCO-512-210-024

Name

intro – Introduction to file formats.

Description

This section outlines the formats of various files. Usually, these structures can be found in the directories **/usr/include** or **/usr/include/sys**.

Name

86rel – Intel 8086 Relocatable Format for Object Modules.

Syntax

#include <sys/relsym86.h>

Description

Intel 8086 Relocatable Format, or *86rel,* is the object module format generated by *masm*(CP), and the input format for the linker *ld*(CP). The include file **relsym86.h** specifies appropriate definitions to access *86rel* format files from C. For the technical details of the *86rel* format, see *Intel 8086 Object Module Format External Product Specification.*

An *86rel* consists of one or more variable length records. Each record has at least three fields: the record type, length, and checksum. The first byte always denotes the record type. There are thirty-one different record types. Only eleven are used by *ld*(CP) and *masm*(CP). The word after the first byte is the length of the record in bytes, exclusive of the first three bytes. Following the length word are typically one or more fields. Each record type has a specific sequence of fields, some of which may be optional or of varying length. The very last byte in each record is a checksum. The checksum byte contains the sum modulo 256 of all other bytes in the record. The sum modulo 256 of all bytes in a record, including the checksum byte, should equal zero.

With few exceptions, *86rel* strings are length prefixed and have no trailing null. The first byte contains a number between 0 and 40, which is the remaining length of the string in bytes. Although the Intel specification limits the character set to upper case letters, digits, and the characters "?", "@", ":", ".", and "_", *masm*(CP) uses the complete ASCII character set.

The Intel Object Module Format (OMF) specification uses the term "index" to mean a positive integer either in the range 0 to 127, or 128 to 32,768. This terminology is retained in this document and elsewhere in the *86rel* literature. An index has one or two bytes. If the first byte has a leading 0 bit, the index is assumed to have only one byte, and the remainder of the byte represents a positive integer between 0 and 127. If the second byte has a leading 1 bit, the index is assumed to take up two bytes, and the remainder of the word represents a positive integer between 128 and 32,768.

Following is a list of record types and the hexadecimal value of their first byte, as defined in **relsym86.h.**

```
#define MRHEADR   0x6e /*rel module header/*
#define MREGINT   0x70 /*register initialization*/
#define MREDATA   0x72 /*explicit (enumerated) data image*/
#define MRIDATA   0x74 /*repeated (iterated) data image*/
#define MOVLDEF   0x76 /*overlay definition*/
#define MENDREC   0x78 /*block or overlay end record*/
#define MBLKDEF   0x7a /*block definition*/
#define MBLKEND   0x7c /*block end*/
#define MDEBSYM   0x7e /*debug symbols*/
#define MTHEADR   0x80 /*module header,
                        *usually first in a rel file*/
#define MLHEADR   0x82 /*link module header*/
#define MPEDATA   0x84 /*absolute data image*/
#define MPIDATA   0x86 /*absolute repeated (iterated)
                        *data image*/
#define MCOMENT   0x88 /*comment record*/
#define MMODEND   0x8a /*module end record*/
#define MEXTDEF   0x8c /*external definition*/
#define MTYPDEF   0x8e /*type definition*/
#define MPUBDEF   0x90 /*public definition*/
#define MLOCSYM   0x92 /*local symbols*/
#define MLINNUM   0x94 /*source line number*/
#define MLNAMES   0x96 /*name list record*/
#define MSEGDEF   0x98 /*segment definition*/
#define MGRPDEF   0x9a /*group definition*/
#define MFIXUPP   0x9c /*fix up previous data image*/
#define MNONE1    0x9e /*none*/
#define MLEDATA   0xa0 /*logical data image*/
#define MLIDATA   0xa2 /*logical repeated (iterated)
                        *data image*/
#define MLIBHED   0xa4 /*library header*/
#define MLIBNAM   0xa6 /*library names record*/
#define MLIBLOC   0xa8 /*library module locations*/
#define MLIBDIC   0xaa /*library dictionary*/
#define M386END   0x86 /*32 bit module end record*/
#define MPUB386   0x91 /*32 bit public definition*/
#define MLOC386   0x93 /*32 bit logical symbols*/
#define MLIN386   0x95 /*32 bit source line number*/
#define MSEG386   0x99 /*32 bit segment definition*/
#define MFIX386   0x9d /*fix up previous 32 bit data image*/
#define MLED386   0xa1 /*32 bit logical data image*/
#define MLID386   0xa3 /*32 bit logical repeated (iterated) data image*/
```

In the following discussion, the salient features of each record type are given. If the record is not used by either *masm*(CP) or *ld*(CP), it is not listed.

THEADR — The record type byte is 0x80. The THEADR record specifies the name of the source module at assembly-time (see Notes). The sole field is the T-MODULE NAME , which contains a length-prefixed string derived from the base name of the source module.

COMENT — The record type byte is 0x88. The COMENT record may contain a remark generated by the compiler system. *mams*(CP) inserts the string "XENIX 8086 ASSEMBLER ."

MODEND — The record type byte is 0x8a. The MODEND record terminates a module. It can specify whether the current module is to be used as the entry point to the linked executable. If the module is an entry point, the MODEND record can then specify the address of the entry point within the executable.

EXTDEF — The record type byte is 0x8c. The EXTDEF record contains the names and types of symbols defined in other modules by a PUBDEF record (see below). This corresponds to the C storage class "extern." The fields consist of one or more length-prefixed strings, each with a following type index. The indices reference a TYPDEF record seen earlier in the module. *masm*(CP) generates only one EXTDEF per exterior symbol.

TYPDEF — The record type byte is 0x8e. The TYPDEF record gives a description of the type (size and storage attributes) of an object or objects. This description can then be referenced by EXTDEF , PUBDEF , and other records.

PUBDEF — The record type byte is 0x90. The PUBDEF record gives a list of one or more names that may be referenced by other modules at link-time ("publics"). The list of names is preceded by a group and segment index, which reference the location of the start of the list of publics within the current segment and group. If the segment and group indices are zero, a frame number is given to provide an absolute address in the module. The list consists of one or more of length-prefixed strings, each associated with a 16-bit offset within the current segment and a type index referring to a TYPDEF .

LNAMES — The record type byte is 0x96. The LNAMES record gives a series of length-prefixed strings which are associated with name indices within the current module. Each name is indexed in sequence given starting with 1. The names may then be referenced within the current module by successive SEGDEF and GRPDEF records to provide strings for segments, classes, overlays or groups.

SEGDEF — The record type byte is 0x98. The SEGDEF record provides an index to reference a segment, and information concerning segment addressing and attributes. This index may be used by other records to refer to the segment. The first word in the record after the length field gives information about the alignment, and about combination attributes of the segment. The next word is the segment length in bytes. Note that this restrains segments to a maximum 645,536 bytes in length. Following this word is an index (see above) for the segment. Lastly, the SEGDEF may optionally contain class and/or overlay index fields.

GRPDEF — The record type is 0x9a. The GRPDEF record provides a name to reference several segments. The group name is implemented as an index (see above).

FIXUPP — The record byte is 0x9c. The FIXUPP record specifies one or more load-time address modifications ("fixups"). Each fixup refers to a location in a preceeding LEDATA (see below) record. The fixup is specified by four data; a location, a mode, a target and a frame. The frame and target may be specified explicitly or by reference to an already defined fixup.

LEDATA — The record type byte is 0xa0. This record provides a contiguous text or data image which the loader *ld*(CP) uses to construct a portion of an 8086 run-time executable. The image might require additional processing (see FIXUPP) before being loaded into the executable. The image is preceeded by two fields, a segment index and an enumerated data offset. The segment index (see INDEX) specifies a segment given by a previously seen SEGDEF . The enumerated data offset (a word) specifies the offset from the start of this segment.

See Also

as(CP), ld(CP)

Notes

If you attempt to load a number of modules assembled under the same basename, the loader will try to put them all in one big segment. In 286 programs, segment size is limited to 64K. In a large program the resulting segment size can easily exceed 64K. A large model code executable results from the link of one or more modules, composed of segments that aggregate into greater than 64K of text.

Hence, be sure that the assembly-time name of the module has the same basename as the source. This can occur if the source module is preprocessed not by *cc*(CP), but, for example, by hand or shell script, prior to assembly. The following example is incorrect:

```
#incorrect
cc -E module1.c |filter > x.c
cc x.c
mv x.o module1.o
cc -E module2.c |filter > x.c
cc x.c
mv x.o module2.o
cc -E module3.c |filter > x.c
cc x.c
mv x.o module3.o
ld module1.o module2.o module3.o
```

To avoid this, each of the modules should have a unique name when assembled, as follows:

```
#correct
cc -E module1.c |filter > x.c
cc -S x.c
mv x.s module1.s
as module1.s
.
.
.
ld module1.o module2.o module3.o
```

Name

a.out – Format of assembler and link editor output.

Description

A.out is the output file of the assembler *masm* and the link editor *ld*. Both programs will make *a.out* executable if there were no errors in assembling or linking, and no unresolved external references.

The format of *a.out*, called the *x.out* or segmented *x.out* format, is defined by the files **/usr/include/a.out.h** and **/usr/include/sys/relsym.h**. The *a.out* file has the following general layout:

1. Header.
2. Extended header.
3. File segment table (for segmented formats).
4. Segments (Text, Data, Symbol, and Relocation).

In the segmented format, there may be several text and data segments, depending on the memory model of the program. Segments within the file begin on boundaries which are multiples of 512 bytes as defined by the file's pagesize.

Format

```
/*
 * The main and extended header structures.
 * For x.out segmented (XE_SEG):
 *     1) fields marked with (s) must contain sums of xs_psize for
 *        non-memory images, or xs_vsize for memory images.
 *     2) the contents of fields marked with (u) are undefined.
 */

struct xexec {                  /* x.out header */
    unsigned short  x_magic;    /* magic number */
    unsigned short  x_ext;      /* size of header extension */
    long      x_text;     /* size of text segment (s) */
    long      x_data;     /* size of initialized data (s) */
    long      x_bss;      /* size of uninitialized data (s) */
    long      x_syms;     /* size of symbol table (s) */
    long      x_reloc;    /* relocation table length (s) */
    long      x_entry;    /* entry point, machine dependent */
```

```
    char        x_cpu;      /* cpu type & byte/word order */
    char        x_relsym;  /* relocation & symbol format (u) */
    unsigned short   x_renv;        /* run-time environment */
};


struct xext {                   /* x.out header extension */
    long        xe_trsize;      /* size of text relocation (s) */
    long        xe_drsize;      /* size of data relocation (s) */
    long        xe_tbase;       /* text relocation base (u) */
    long        xe_dbase;       /* data relocation base (u) */
    long        xe_stksize;     /* stack size (if XE_FS set) */
                /* the following must be present if XE_SEG */
    long        xe_segpos;      /* segment table position */
    long        xe_segsize;     /* segment table size */
    long        xe_mdtpos;      /* machine dependent table position */
    long        xe_mdtsize;     /* machine dependent table size */
    char        xe_mdttype;     /* machine dependent table type */
    char        xe_pagesize;    /* file pagesize, in multiples of 512 */
    char        xe_ostype;      /* operating system type */
    char        xe_osvers;      /* operating system version */
    unsigned short   xe_eseg;   /* entry segment, machine dependent */
    unsigned short   xe_sres;   /* reserved */
};


struct xseg {               /* x.out segment table entry */
    unsigned short   xs_type;   /* segment type */
    unsigned short   xs_attr;   /* segment attributes */
    unsigned short   xs_seg;    /* segment number */
    char        xs_align;       /* log base 2 of alignment */
    char        xs_cres;        /* unused */
    long        xs_filpos;      /* file position */
    long        xs_psize;       /* physical size (in file) */
    long        xs_vsize;       /* virtual size (in core) */
    long        xs_rbase;       /* relocation base address/offset */
    unsigned short   xs_noff;   /* segment name string table offset */
    unsigned short   xs_sres;   /* unused */
    long        xs_lres;        /* unused */
};


struct xiter {              /* x.out iteration record */
    long        xi_size;        /* source byte count */
    long        xi_rep;     /* replication count */
    long        xi_offset;  /* destination offset in segment */
};
```

```
struct xlist {                  /* xlist structure for xlist(3). */
    unsigned short    xl_type;     /* symbol type */
    unsigned short    xl_seg;      /* file segment table index */
    long        xl_value;          /* symbol value */
    char        *xl_name;          /* pointer to asciz name */
};

struct aexec {                  /* a.out header */
    unsigned short    xa_magic;     /* magic number */
    unsigned short    xa_text;      /* size of text segment */
    unsigned short    xa_data;      /* size of initialized data */
    unsigned short    xa_bss;       /* size of unitialized data */
    unsigned short    xa_syms;      /* size of symbol table */
    unsigned short    xa_entry;     /* entry point */
    unsigned short    xa_unused;    /* not used */
    unsigned short    xa_flag;      /* relocation info stripped */
};


struct nlist {                  /* nlist structure for nlist(3). */
    char        n_name[8];     /* symbol name */
    int     n_type;            /* type flag */
    unsigned    n_value;       /* value */
};


struct bexec {          /* b.out header */
    long    xb_magic; /* magic number */
    long    xb_text;  /* text segment size */
    long    xb_data;  /* data segment size */
    long    xb_bss;   /* bss size */
    long    xb_syms;  /* symbol table size */
    long    xb_trsize; /* text relccation table size */
    long    xb_drsize; /* data relocation table size */
    long    xb_entry; /* entry point */
};
```

See Also

masm(CP), ld(CP), nm(CP), strip(CP), xlist(S).

Name

acct – Format of per-process accounting file.

Description

Files produced as a result of calling *acct*(S) have records in the form defined by <sys/acct.h>.

In *ac_flag*, the AFORK flag is turned on by each *fork*(S) and turned off by an *exec*(S). The *ac_comm* field is inherited from the parent process and is reset by any *exec*. Each time the system charges the process with a clock tick, it also adds the current process size to *ac_mem* computed as follows:

(data size) + (text size) / (number of in-core processes using text)

The value of *ac_mem/ac_stime* can be viewed as an approximation to the mean process size, as modified by text-sharing.

See Also

acct(C), acctcom(C), acct(S)

Notes

The *ac_mem* value for a short-lived command gives little information about the actual size of the command, because *ac_mem* may be incremented while a different command (e.g., the shell) is being executed by the process.

Name

ar – Archive file format.

Description

The archive command *ar* is used to combine several files into one. Archives are used mainly as libraries to be searched by the link editor *ld*(C).

A file produced by *ar* has a magic number at the start, followed by the constituent files, each preceded by a file header. The magic number is 0177545 octal (or 0xff65 hexadecimal). The header of each file is declared in **/usr/include/ar.h**.

Each file begins on a word boundary; a null byte is inserted between files if necessary. Nevertheless the size given reflects the actual size of the file exclusive of padding.

Notice there is no provision for empty areas in an archive file.

See Also

ar(CP), ld(CP)

Name

archive – Default backup device information.

Description

/etc/default/archive contains information on system default backup devices for use by *sysadmin*(C). The device entries are in the following format:

name=value [name=value] ...

value may contain white spaces if quoted, and newlines may be escaped with a backslash.

The following names are defined for */etc/default/archive*:

bdev	Name of the block interface device.
cdev	Name of the character interface device.
size	Size of the volume in either blocks or feet.
density	Volume density, such as 1600. If this value is missing or null, then *size* is in blocks; otherwise the *size* is in feet.
format	Command used to format the archive device.
blocking	Blocking factor.
desc	A description of the device, such as "Cartridge Tape."

See Also

sysadmin(C)

Name

backup – Incremental dump tape format.

Description

The *backup* and *restore* commands are used to write and read incremental dump magnetic tapes.

The backup tape consists of a header record, some bit mask records, a group of records describing file system directories, a group of records describing file system files, and some records describing a second bit mask.

The header record and the first record of each description have the format described by the structure included by:

#include <dumprestor.h>

Fields in the *dumprestor* structure are described below.

NTREC is the number of 512 byte blocks in a physical tape record. MLEN is the number of bits in a bit map word. MSIZ is the number of bit map words.

The TS_ entries are used in the *c_type* field to indicate what sort of header this is. The types and their meanings are as follows:

TS_TYPE	Tape volume label.
TS_INODE	A file or directory follows. The *c_dinode* field is a copy of the disk inode and contains bits telling what sort of file this is.
TS_BITS	A bit mask follows. This bit mask has one bit for each inode that was backed up.
TS_ADDR	A subblock to a file (*TS_INODE*). See the description of *c_count* below.
TS_END	End of tape record.
TS_CLRI	A bit mask follows. This bit mask contains one bit for all inodes that were empty on the file system when backed up.
MAGIC	All header blocks have this number in *c_magic*.
CHECKSUM	Header blocks checksum to this value.

The fields of the header structure are as follows:

c_type The type of the header.

c_date The date the backup was taken.

c_ddate The date the file system was backed up.

c_volume The current volume number of the backup.

c_tapea The current block number of this record. This is counting 512 byte blocks.

c_inumber The number of the inode being backed up if this is of type TS_INODE.

c_magic This contains the value MAGIC above, truncated as needed.

c_checksum This contains whatever value is needed to make the block sum to CHECKSUM.

c_dinode This is a copy of the inode as it appears on the file system.

c_count The following count of characters describes the file. A character is zero if the block associated with that character was not present on the file system; otherwise, the character is nonzero. If the block was not present on the file system no block was backed up and it is replaced as a hole in the file. If there is not sufficient space in this block to describe all of the blocks in a file, TS_ADDR blocks will be scattered through the file, each one picking up where the last left off.

c_addr This is the array of characters that is used as described above.

Each volume except the last ends with a tapemark (read as an end of file). The last volume ends with a TS_END block and then the tapemark.

The structure *idates* describes an entry of the file where backup history is kept.

See Also

backup(C), restore(C), filesystem(F)

Name

checklist – List of file systems processed by *fsck*.

Description

The **/etc/checklist** file contains a list of the file systems to be checked when *fsck*(C) is invoked without arguments. The list contains at most 15 **special file** names. Each **special file** name must be on a separate line and must correspond to a file system.

See Also

fsck(C)

Name

core – Format of core image file.

Description

XENIX writes out a core image of a terminated process when any of various errors occur. See *signal*(S) for the list of reasons; the most common are memory violations, illegal instructions, bus errors, and user-generated quit signals. The core image is called *core* and is written in the process' working directory (provided it can be; normal access controls apply). A process with an effective user ID different from the real user ID will not produce a core image.

The first section of the core image is a copy of the system's per-user data for the process, including the registers as they were at the time of the fault. The size of this section depends on the parameter *usize*, which is defined in **/usr/include/sys/param.h**. The remainder represents the actual contents of the user's core area when the core image was written. If the text segment is read-only and shared, or separated from data space, it is not dumped.

The format of the information in the first section is described by the *user* structure of the system, defined in **/usr/include/sys/user.h**. The locations of registers, are outlined in **/usr/include/sys/reg.h**.

See Also

adb(CP), setuid(S), signal(S)

Name

cpio – Format of cpio archive.

Description

The *header* structure, when the **c** option is not used, is:

```
struct {
        short   h_magic,
                h_dev,
                h_ino,
                h_mode,
                h_uid,
                h_gid,
                h_nlink,
                h_rdev,
                h_mtime[2],
                h_namesize,
                h_filesize[2];
        char    h_name[h_namesize rounded to word];
} Hdr;
```

When the **c** option is used, the *header* information is described by the statement below:

```
sscanf(Chdr,"%6o%6o%6o%6o%6o%6o%6o%6o%11lo%6o%6o%s",
        &Hdr.h_magic,&Hdr.h_dev,&Hdr.h_ino,&Hdr.h_mode,
        &Hdr.h_uid,&Hdr.h_gid,&Hdr.h_nlink,&Hdr.h_rdev,
        &Longtime,&Hdr.h_namesize,&Longfile,Hdr.h_name);
```

Longtime and *Longfile* are equivalent to *Hdr.h_mtime* and *Hdr.h_filesize*, respectively. The contents of each file is recorded in an element of the array of varying length structures, *archive*, together with other items describing the file. Every instance of *h_magic* contains the constant 070707 (octal). The items *h_dev* through *h_mtime* have meanings explained in *stat*(S). The length of the null-terminated pathname *h_name*, including the null byte, is given by *h_namesize*.

The last record of the *archive* always contains the name TRAILER!!!. Special files, directories, and the trailer are recorded with *h_filesize* equal to zero.

See Also

cpio(C), find(C), stat(S)

Name

dir – Format of a directory.

Syntax

#include <sys/dir.h>

Description

A directory behaves exactly like an ordinary file, except that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its inode entry (see *filesystem*(F)). The structure of a directory is given in the include file **/usr/include/sys/dir.h**.

By convention, the first two entries in each directory are"dot" (.) and "dotdot" (..). The first is an entry for the directory itself. The second is for the parent directory. The meaning of dotdot is modified for the root directory of the master file system; there is no parent, so dotdot has the same meaning as dot.

See Also

filesystem(F)

Name

dump – Incremental dump tape format.

Description

The *dump* and *restor* commands are used to write and read incremental dump magnetic tapes.

The dump tape consists of a header record, some bit mask records, a group of records describing file system directories, a group of records describing file system files, and some records describing a second bit mask.

The header record and the first record of each description have the format described by the structure included by:

#include <dumprestor.h>

Fields in the *dumprestor* structure are described below.

NTREC is the number of 512 byte blocks in a physical tape record. MLEN is the number of bits in a bit map word. MSIZ is the number of bit map words.

The TS_ entries are used in the *c_type* field to indicate what sort of header this is. The types and their meanings are as follows:

TS_TYPE	Tape volume label.
TS_INODE	A file or directory follows. The *c_dinode* field is a copy of the disk inode and contains bits telling what sort of file this is.
TS_BITS	A bit mask follows. This bit mask has a one-bit for each inode that was dumped.
TS_ADDR	A subblock to a file (*TS_INODE*). See the description of *c_count* below.
TS_END	End of tape record.
TS_CLRI	A bit mask follows. This bit mask contains a one-bit for all inodes that were empty on the file system when dumped.
MAGIC	All header blocks have this number in *c_magic*.
CHECKSUM	Header blocks checksum to this value.

The fields of the header structure are as follows:

c_type The type of the header.

c_date The date the dump was taken.

c_ddate The date the file system was dumped from.

c_volume The current volume number of the dump.

c_tapea The current block number of this record. This is counting 512 byte blocks.

c_inumber The number of the inode being dumped if this is of type TS_INODE.

c_magic This contains the value MAGIC above, truncated as needed.

c_checksum This contains whatever value is needed to make the block sum to CHECKSUM.

c_dinode This is a copy of the inode as it appears on the file system.

c_count This is the count of characters following that describe the file. A character is zero if the block associated with that character was not present on the file system, otherwise the character is nonzero. If the block was not present on the file system no block was dumped and it is replaced as a hole in the file. If there is not sufficient space in this block to describe all of the blocks in a file, TS_ADDR blocks will be scattered through the file, each one picking up where the last left off.

c_addr This is the array of characters that is used as described above.

Each volume except the last ends with a tapemark (read as an end of file). The last volume ends with a TS_END block and then the tapemark.

The structure *idates* describes an entry of the file where dump history is kept.

See Also

dump(C), restor(C), filesystem(F)

Name

filesys – Default information for mounting filesystems.

Description

/etc/default/filesys contains information for mounting filesystems in the following format:

name=value [name=value] ...

value may contain white spaces if quoted, and newlines may be escaped with a backslash.

mnt (see *mount*(C)) and *sysadmin*(C) use the information in the */etc/default/filesys* when the system comes up multiuser. The following names are defined for */etc/default/filesys*:

bdev	Name of the block interface device.
cdev	Name of the character interface device.
size	Size in blocks.
mountdir	Directory on which the filesystem is mounted.
desc	A description of the filesystem. For example, "User filesystem."
mountflags	Any flags passed to the **mount**(C) command.
fsckflags	Any flags passed to the **fsck**(C) command.
rcmount	Whether or not to mount the filesystem when the system goes multiuser. Can be "yes", "no" or "prompt". If set to "prompt", you are prompted when it is time to mount the filesystem.

See Also

mount(C), sysadmin(C)

Name

file system – Format of a system volume.

Syntax

#include <sys/filsys.h>
#include <sys/types.h>
#include <sys/param.h>

Description

Every file system storage volume (for example, a hard disk) has a common format for certain vital information. Every such volume is divided into a certain number of 256 word (512 byte) blocks. Block 0 is unused and is available to contain a bootstrap program or other information.

Block 1 is the *super-block*. The format of a super-block is described in **/usr/include/sys/filesys.h**. In that include file, *S_isize* is the address of the first data block after the i-list. The i-list starts just after the super-block in block 2; thus the i-list is *s_isize*−2 blocks long. *S_fsize* is the first block not potentially available for allocation to a file. These numbers are used by the system to check for bad block numbers. If an "impossible" block number is allocated from the free list or is freed, a diagnostic is written on the console. Moreover, the free array is cleared so as to prevent further allocation from a presumably corrupted free list.

The free list for each volume is maintained as follows. The *s_free* array contains, in *s_free*[1], . . ., *s_free*[*s_nfree*−1], up to 49 numbers of free blocks. *S_free*[0] is the block number of the head of a chain of blocks constituting the free list. The first long in each free-chain block is the number (up to 50) of free-block numbers listed in the next 50 longs of this chain member. The first of these 50 blocks is the link to the next member of the chain. To allocate a block: decrement *s_nfree*, and the new block is *s_free*[*s_nfree*]. If the new block number is 0, there are no blocks left, so give an error. If *s_nfree* becomes 0, read in the block named by the new block number, replace *s_nfree* by its first word, and copy the block numbers in the next 50 longs into the *s_free* array. To free a block, check if *s_nfree* is 50; if so, copy *s_nfree* and the *s_free* array into it, write it out, and set *s_nfree* to 0. In any event set *s_free*[*s_nfree*] to the freed block's number and increment *s_nfree*.

S_tfree is the total free blocks available in the file system.

S_ninode is the number of free i-numbers in the *s_inode* array. To allocate an inode: if *s_ninode* is greater than 0, decrement it and return *s_inode*[*s_ninode*]. If it was 0, read the i-list and place the numbers of all free inodes (up to 100) into the *s_inode* array, then try again. To free an inode, provided *s_ninode* is less than 100, place its number into *s_inode*[*s_ninode*] and increment *s_ninode*. If *s_ninode* is already 100, do not bother to enter the freed inode into any table. This list of inodes only speeds up the allocation process. The information about whether the inode is really free is maintained in the inode itself.

S_tinode is the total free inodes available in the file system.

S_flock and *s_ilock* are flags maintained in the core copy of the file system while it is mounted and their values on disk are immaterial. The value of *s_fmod* on disk is also immaterial, and is used as a flag to indicate that the super-block has changed and should be copied to the disk during the next periodic update of file system information.

S_ronly is a read-only flag to indicate write-protection.

S_time is the last time the super-block of the file system was changed, and is a double precision representation of the number of seconds that have elapsed since 00:00 Jan. 1, 1970 (GMT). During a reboot, the *s_time* of the super-block for the root file system is used to set the system's idea of the time.

I-numbers begin at 1, and the storage for inodes begins in block 2. Also, inodes are 64 bytes long, so 8 of them fit into a block. Therefore, inode i is located in block $(i+15)/8$, and begins $64\times((i+15) \pmod 8)$ bytes from its start. Inode 1 is reserved for future use. Inode 2 is reserved for the root directory of the file system, but no other i-number has a built-in meaning. Each inode represents one file. For the format of an inode and its flags, see *inode*(F).

Files

/usr/include/sys/filsys.h

/usr/include/sys/stat.h

See Also

fsck(C), mkfs(C), inode(F)

Name

fstab – File system mount and check commands.

Description

fstab is an ASCII text file containing information that is passed to the *mount*(C) and *fsck*(C) commands that are executed from **/etc/rc**. A typical **/etc/fstab** file might look like this:

```
# device        directory    optional flags
/dev/u          /u           fsckflags="-y -D"
/dev/archive    /archive     mountflags="-r" fsckflags="-f"
```

The first column lists the device to be mounted and the second column gives the mount point (directory) for the device.

The third column lists any optional flags. Optional flags are:

fsckflags	–	Flags that are passed to *fsck*.
mountflags	–	Flags that are passed to *mount*.
prompt	–	If set to "y", prompts whether or not to mount filesystem. Default is "n".

Comment lines start with a number sign (#).

See Also

fsck(C), mount(C)

Name

gettydefs – Speed and terminal settings used by getty.

Description

The **/etc/gettydefs** file contains information used by *getty*(M) to set up the speed and terminal settings for a line. It supplies information on what the *login* prompt should look like. It also supplies the speed to try next if the user indicates the current speed is not correct by typing a BREAK character.

Each entry in **/etc/gettydefs** has the following format:

label# initial-flags # final-flags # login-prompt #next-label [# login-program]

Each entry is followed by a blank line. The various fields can contain quoted characters of the form **\b, \n, \c**, etc., as well as *\nnn*, where *nnn* is the octal value of the desired character. The various fields are:

label — Identifies the **/etc/gettydefs** entry to *getty*. This could be a letter or number. The label corresponds to the line mode field in **/etc/ttys**. *Init* passes the line mode as an argument to *getty*.

initial-flags — Sets the initial *ioctl*(S) settings if a terminal type is not specified to *getty*. The flags that *getty* understands are the same as the ones listed in *tty*(M). Normally only the speed flag is required in the *initial-flags*. *Getty* automatically sets the terminal to raw input mode and takes care of most of the other flags. The *initial-flag* settings remain in effect until *getty* executes *login*(M).

final-flags — Sets the same values as the *initial-flags*. These flags are set just prior to *getty* executing *login-program*. The speed flag is again required. The composite flag **SANE** takes care of most of the other flags that need to be set so that the processor and terminal are communicating in a rational fashion. The other two commonly specified *final-flags* are **TAB3**, so that tabs are sent to the terminal as spaces, and **HUPCL**, so that the line is hung up on the final close.

login-prompt — Contains login prompt message that greets users. Unlike the above fields where white space is ignored (a space, tab, or new-line), it is included in the *login-prompt* field. The '@' in the login-prompt field is expanded to the first line in **/etc/systemid** (unless the '@' is preceded by a '\'). Several character sequences are recognized, including:

\n	Linefeed
\r	Carriage return
\v	Vertical tab
nnn	(3 octal digits) Specify ASCII character
\t	Tab
\f	Form feed
\b	Backspace

next-label Identifies the next entry in *gettydefs* for *getty* to try if the current one is not successful. *Getty* tries the next label if a user presses the BREAK key while attempting to log in to the system. Groups of entries, for example, for dial-up lines or for TTY lines, should form a closed set so that *getty* cycles back to the original entry if none of the entries is successful. For instance, **2400** linked to **1200**, which in turn is linked to **300**, which finally is linked to **2400**.

login-program The name of the program that actually logs the user onto XENIX. The default program is **/etc/login**. If preceded by the keyword **AUTO**, *getty* will not prompt for a username, but instead uses its first argument as the username and executes the *login-program* immediately.

If *getty* is called without a second argument, then the first entry of **/etc/gettydefs** is used, thus making the first entry of **/etc/gettydefs** the default entry. The first entry is also used if *getty* can not find the specified *label*. If **/etc/gettydefs** itself is missing, there is one entry built into the command which will bring up a terminal at **300** baud.

After modifying **/etc/gettydefs**, run it through *getty* with the check option to be sure there are no errors.

Files

/etc/gettydefs

See Also

ioctl(S), getty(M), login(M)

Name

inittab – Alternative login terminals file.

Description

telinit(C) reads *inittab* and converts it into a *ttys*(M)-format file. *init*(M) reads **/etc/ttys** to determine for which terminals logins are allowed.

Each line in *inittab* has the form:

id:*run-levels*:*action*:**/etc/getty** *tty mode*

id A one- to four-character name that uniquely identifies this line. It is recommended that if *tty* is **tty***xx* that the *id* then be "*xx*".

run-levels
A list of digits ranging from **0** to **6**. This list specifies which *telinit* states are concerned with this line. If the *run-levels* list is empty, then it is assumed to be "**0123456**" (all states).

action
Whether or not logins are allowed on *tty*:

off
Logins are not allowed in any of the listed *run-levels*.

respawn
Logins are allowed only in the listed *run-levels*.

ondemand
Identical to "**respawn**".

tty The filename of a character device special file. Only the filename is supplied; the path is assumed to be **/dev**.

mode
A single character supplied as an argument to the *getty*(M) program. It defines the line characteristics (such as the baud rate) for the terminal, and must match one of the names listed in **/etc/gettydefs**.

Exactly one space must separate *ttys* from ...:**/etc/getty** and from *mode*. No other spaces or tabs are allowed.

Files

/etc/inittab

See Also

disable(C), enable(C), init(M), getty(M), gettydefs(F), telinit(C), ttys(M)

Notes

inittab is provided for users more familiar with the *telinit* approach to terminal administration, as opposed to the standard XENIX *enable*(C)/*disable*(C) approach. It is intended that a full integration of these two approaches will be provided in a future version of XENIX.

Name

inode – Format of an inode.

Syntax

#include <sys/types.h>
#include <sys/ino.h>

Description

An inode for a plain file or directory in a file system has the structure defined by **<sys/ino.h>**. For the meaning of the defined types *off_t* and *time_t* see *types*(F).

Files

/usr/include/sys/ino.h

See Also

stat(S), filesystem(F), types(F)

Name

mapchan – Format of tty device mapping files.

Description

mapchan configures the mapping of information input and output of XENIX.

Each unique *channel* map requires 1024 bytes (a 1K buffer) for mapping the input and output of characters. No maps are required if no *channels* are mapped.

A method of sharing maps is implemented for *channels* that have the same map in place. Each additional, unique map allocates an additional buffer. The maximum number of map buffers available on a system is configured in the kernel, and is adjustable via the link kit (see *config*(C) and *configure*(C)). Buffers of maps no longer in use are returned for use by other maps.

Example of a Map File

The internal character set used by XENIX is defined by the right column of the input map, and the first column of the output map in place on that line. By default, this is the 8-bit ASCII character set which is also known as the dpANS X3.4.2 and ISO/TC97/SC2 or ISO 8859 Level I character sets. It supports the Latin alphabet and can represent most European languages.

Any character value not given is assumed to be a straight mapping, only the differences are shown in the *mapfile*. The left hand columns must be unique. More than one occurence of any entry is an error. Right hand column characters can appear more than once. This is "many to one" mapping. Nulls can be produced with dead or compose sequences or as part of an output string.

It is recommended that no mapping be enabled on the *channel* used to create or modify the mapping files. This prevents any confusion of the actual values being entered due to mapping. It is also recommended that numeric rather than character representations be used in most cases, as these are not likely to be subject to mapping. Use comments to identify the characters represented. Refer to the *ascii*(M) manual page and the hardware reference manual for the device being mapped for the values to assign.

```
#
# sharp/pound/cross-hatch is the comment character
# however, a quoted # ('#') is 0x23, not a comment
#
# beep, input, output, dead and compose are special
# keywords and should appear as shown.
#

beep            # sound the bell when errors occur

input
a b
c d

dead p
q r             # p followed by q yields r.
s t             # p followed by s yields t.

dead u
v w             # u followed by v yields w.

compose x       # x is the compose key (only one allowed).
y z A           # x followed by y and z yields A.
B C D           # x followed by B and C yields D.

output
e f             # e is mapped to f.
g h i j         # g is mapped to hij - one to many.
k l m n o       # k is mapped to lmno.
```

All of the single letters above can be in one of these formats:

```
56              # decimal
045             # octal
0xfa            # hexadecimal
'b'             # quoted char
'\076'          # quoted octal
'\x4a'          # quoted hex
```

All of the above formats are translated to single byte values.

Diagnostics

mapchan performs these error checks when processing the mapfile:

More than one compose key.
Characters mapped to more than one thing.
Syntax errors in the byte values.
Missing input or output keywords.
Dead or compose keys also occuring in the input section.
Extra information on a line.
Mapping a character to null.

Characters are displayed as the 7-bit value instead of the 8-bit value. Use **stty −a** to verify that **-istrip** is set. Make sure **input** is mapping to the 8859 character set, **output** is mapping from the 8859 to the device display character set. **dead** and **compose** sequences are **input** mapping and should be going to 8859.

Files

/etc/default/mapchan
/usr/lib/mapchan/map.stdrom
/usr/lib/mapchan/map.*

See Also

ascii(M), keyboard(HW), lp(C), lpadmin(C), mapchan(M), mapkey(M), parallel(HW), screen(HW), serial(HW), setkey(M), tty(M)

Notes

Some foreign keyboards and display devices do not contain characters commonly used by XENIX command shells and the C programming language. Do not attempt to use such devices for system administration tasks.

Not all terminals or printers can display all the characters that can be represented using this utility. Refer to the device's hardware manual for information on the capabilities of the peripheral device.

WARNING: Use of mapping files that specify a different "internal" character set per-channel, or a set other than the 8-bit ASCII set supplied by default can cause strange side effects. It is especially important to retain the 7-bit ASCII portion of the character set (see *ascii*(M)). XENIX utilities and applications assume these values. Media transported between machines with different internal code set mappings may not be portable as no mapping is performed on block devices, such as tape and floppy drives. *mapchan* can be used to "translate" from one internal character set to another.

Do not set ISTRIP (see *stty*(C)) on channels that have mapping that includes eight bit characters.

Name

master – Master device information table.

Description

master contains device information used by *config*(C) to generate the configuration files. The file consists of 4 parts, each separated by a line with a dollar sign ($) in column 1.

- Part 1 contains device information.
- Part 2 contains the line discipline table.
- Part 3 contains names of devices that have aliases.
- Part 4 contains tunable parameter information.

Any line with an asterisk (*) in column 1 is treated as a comment.

Part 1

This part contains definitions for the system devices. Each line has 14 fields with the fields delimited by tabs and/or blanks:

Field 1: Device name (8 chars. maximum).
Field 2: Number of interrupt vectors.
Field 3: Device mask (octal). Each "on" bit indicates that the driver has the corresponding handler or structure:

000400 tty structure.
000200 Not used.
000100 Initialization handler.
000040 Clock time poll routine.
000020 Open handler.
000010 Close handler.
000004 Read handler.
000002 Write handler.
000001 Ioctl handler.

The clock time poll routine, if present in the driver, is called every clock tick in which the clock interrupted task-time processing.

Field 4: Device type indicator (octal):

000200 Not used
000100 No qswtch on interrupt.
000040 Not used.
000020 Required device.
000010 Block device.
000004 Character device.
000002 Not used.
000001 Not used.

Field 5:	Handler prefix (4 chars. maximum). Usually same as Field 1. The routines of **dev.c** should begin *dev*... The tty structure of **dev.c** should be named *dev_tty*.
Field 6:	Not used.
Field 7:	Major device number for block-type device.
Field 8:	Major device number for character-type device.
Field 9:	Maximum number of devices per controller.
Field 10:	The *spl* level (1 - 7) at which the device's interrupt routine should be called.
Fields 11-14:	Maximum of four interrupt vector addresses (octal). Each address is followed by a unique letter or a blank.

Devices that are not interrupt-driven have an interrupt vector size of zero. Devices that generate interrupts but are not of the standard character or block device mold, should be specified with a type (field 4) which has neither the block nor character bits set.

Part 2

This part contains definitions for the system line discipline. Each line has 9 fields. Each field is a maximum of 8 characters delimited by a blank if less than 8:

Field 1:	Device associated with this line.
Field 2:	Open routine.
Field 3:	Close routine.
Field 4:	Read routine.
Field 5:	Write routine.
Field 6:	Ioctl routine.
Field 7:	Receiver interrupt routine.
Field 8:	Transmitter interrupt routine.
Field 9:	Modem control interrupt routine.

Part 3

This part contains definitions for device aliases. Each line has 2 fields:

Field 1:	Alias name of device (8 chars. maximum).
Field 2:	Reference name of device as given in part 1 (8 chars. maximum).

Aliases may be used in place of actual device names when creating the *config*(C) description file.

Part 4

This part contains the names and default values for tunable parameters. Each line has 2 or 3 fields:

Field 1:	Parameter name to be used in the *config*(C) description file (20 chars. maximum).
Field 2:	Parameter name as it will appear in the resulting **c.c** file (20 chars. maximum).
Field 3:	Default parameter value (20 chars. maximum).

If a parameter has no default value, an explicit specification for the parameter must be given in the description file. See *config*(C) for a list of the tunable parameters.

See Also

config(C), configure(C)

Name

mnttab – Format of mounted file system table.

Syntax

#include <stdio.h>
#include <mnttab.h>

Description

The **/etc/mnttab** file contains a table of devices mounted by the *mount*(C) command.

Each table entry contains the pathname of the directory on which the device is mounted, the name of the device special file, the read/write permissions of the special file, and the date on which the device was mounted.

The maximum number of entries in *mnttab* is based on the system parameter NMOUNT located in **/usr/sys/conf/space.c**, which defines the number of allowable mounted special files.

See Also

mount(C)

Name

sccsfile – Format of an SCCS file.

Description

An SCCS file is an ASCII file. It consists of six logical parts: the *checksum*, the *delta table* (contains information about each delta), *user names* (contains login names and/or numerical group IDs of users who may add deltas), *flags* (contains definitions of internal keywords), *comments* (contains arbitrary descriptive information about the file), and the *body* (contains the actual text lines intermixed with control lines). Each logical part of an SCCS file is described in detail below.

Throughout an SCCS file there are lines which begin with the ASCII SOH (start of heading) character (octal 001). This character is hereafter referred to as *the control character* and will be represented graphically as @. Any line described below which is not depicted as beginning with the control character is prevented from beginning with the control character. Entries of the form DDDDD represent a five digit string (a number between 00000 and 99999).

Checksum

The checksum is the first line of an SCCS file. The form of the line is:

```
@hDDDDD
```

The value of the checksum is the sum of all characters, except those of the first line. The @hR provides a *magic number* of (octal) 064001.

Delta Table

The delta table consists of a variable number of entries of the form:

```
@s DDDDD/DDDDD/DDDDD
@d <type> <SCCS ID> yr/mo/da hr:mi:se <pgmr> DDDDD DD
@i DDDDD ...
@x DDDDD ...
@g DDDDD ...
@m <MR number>
 .
 .
@c <comments> ...
 .
 .
@e
```

The first line (@s) contains the number of lines inserted/deleted/unchanged respectively. The second line (@d) contains the type of the delta (currently, normal: D, and removed: R), the SCCS ID of the delta, the date and time of creation of the delta, the login name corresponding to the real user ID at the time the delta was created, and the serial numbers of the delta and its predecessor, respectively.

The @i, @x, and @g lines contain the serial numbers of deltas included, excluded, and ignored, respectively. These lines are optional.

The @m lines (optional) each contain one MR number associated with the delta; the @c lines contain comments associated with the delta.

The @e line ends the delta table entry.

User Names

The list of login names and/or numerical group IDs of users who may add deltas to the file, separated by new-lines. The lines containing these login names and/or numerical group IDs are surrounded by the bracketing lines @u and @U. An empty list allows anyone to make a delta.

Flags

Keywords used internally (see *admin*(CP) for more information on their use). Each flag line takes the form:

```
@f <flag>       <optional text>
```

The following flags are defined:

```
@f t    <type of program>
@f v    <program name>
@f i
@f b
@f m    <module name>
@f f    <floor>
@f c    <ceiling>
@f d    <default-sid>
@f n
@f j
@f l    <lock-releases>
@f q    <user defined>
```

The **t** flag defines the replacement for the identification keyword. The **v** flag controls prompting for MR numbers in addition to comments; if the optional text is present it defines an MR number validity checking program. The **i** flag controls the warning/error aspect of the "No id keywords" message. When the **i** flag is not present, this message is only a warning; when the **i** flag is present, this message will cause a "fatal" error (the file will not be gotten, or the delta will not be made). When the **b** flag is present the **−b** option may be used with the *get* command to cause a branch in the delta tree. The **m** flag defines the first choice for the replacement text of the sccsfile.F identification keyword. The **f** flag defines the "floor" release; the release below which no deltas may be added. The **c** flag defines the "ceiling" release; the release above which no deltas may be added. The **d** flag defines the default SID to be used when none is specified on a *get* command. The **n** flag causes *delta* to insert a "null" delta (a delta that applies *no* changes) in those releases that are skipped when a delta is made in a *new* release (e.g., when delta 5.1 is made after delta 2.7, releases 3 and 4 are skipped). The absence of the **n** flag causes skipped releases to be completely empty. The **j** flag causes *get* to allow concurrent edits of the same base SID. The **l** flag defines a *list* of releases that are *locked* against editing (*get*(CP) with the **−e** option). The **q** flag defines the replacement for the identification keyword.

Comments

Arbitrary text surrounded by the bracketing lines @t and @T. The comments section typically contains a description of the file's purpose.

Body

The body consists of text lines and control lines. Text lines don't begin with the control character, control lines do. There are three kinds of control lines: *insert, delete*, and *end*, as follows:

@I DDDDD
@D DDDDD
@E DDDDD

The digit string (DDDDD) is the serial number corresponding to the delta for the control line.

See Also

admin(CP), delta(CP), get(CP), prs(CP)

XENIX *Programmer's Guide*

Name

stat – Data returned by stat system call.

Syntax

#include <sys/stat.h>

Description

The **sys/stat.h** include file contains the definition for the structure returned by the *stat* and *fstat* functions. The structure is defined as:

```
struct stat{
    dev_t     st_dev;      /*


    ino_t     st_ino;      /* inode number */
    ushort    sh_mode;     /* file mode */
    short     st_nlink;    /* # of links */
    ushort    st_uid;      /* owner uid */
    ushort    st_gid;      /* owner gid */
    dev_t     st_rdev;     /*


    off_t     st_size;     /* file size in bytes */
    time_t    st_atime;    /* time of last access */
    time_t    st_mtime;    /* time of last data modification */
    time_t    st_ctime;    /* time of last file status 'change' */
};
```

Note that the *st_atime*, *st_mtime*, and *st_ctime* values are measured in seconds since 00:00:00 (GMT) on January 1, 1970.

The *st_mode* value is actually a combination of one or more of the following file mode values:

```
S_IFMT     0170000   /* type of file */
S_IFDIR    0040000   /* directory */
S_IFCHR    0020000   /* character special */
S_IFBLK    0060000   /* block special */
S_IFREG    0100000   /* regular */
S_IFIFO    0010000   /* fifo */
S_IFNAM    0050000   /* name special entry */
S_INSEM    01        /* semaphore */
S_INSHD    02        /* shared memory */
S_ISUID    04000     /* set user id on execution */
S_IGUID    02000     /* set group id on execution */
S_ISVTX    01000     /* save swapped text even after use */
S_IREAD    00400     /* read permission, owner */
```

S_IWRITE	00200	/* write permission, owner */
S_IEXEC	00100	/* execute/search permission, owner */

Files

/usr/include/sys/stat.h

See Also

stat(S)

Name

tar – archive format

Description

The command *tar*(C) dumps files to and extracts files from backup media or the hard disk.

Each file is archived in contiguous blocks, the first block being occupied by a header, whose format is given below, and the subsequent blocks of the files occupying the following blocks. All headers and file data start on 512 byte block boundaries and any spare unused space is padded with garbage. The format of a header block is as follows:

```
#define TBLOCK 512
#define NBLOCK 20
#define NAMSIZ 100
union hblock {
    char dummy[TBLOCK];
    struct header {
        char name[NAMSIZ];
        char mode[8];
        char uid[8];
        char gid[8];
        char size[12];
        char mtime[12];
        char chksum[8];
        char linkflag;
        char linkname[NAMSIZ];
        char extno[4];
        char extotal[4];
        char efsize[12];
    } dbuf;
} dblock;
```

The name entry is the path name of the file when archived. If the pathname starts with a zero word, the entry is empty. It is at most 100 bytes long and ends in a null byte. Mode, uid, gid, size, and time modified are the same as described under i-nodes (refer to *filesystem*(F)). The checksum entry has a value such that the sum of the words of the directory entry is zero.

If the entry corresponds to a link, then *linkname* contains the pathname of the file to which this entry is linked and *linkflag* gives a count of the links. No data is put in the archive file.

See Also

filesystem(F), tar(C)

Name

term – Terminal driving tables for nroff.

Description

nroff(CT) uses driving tables to customize its output for various types of output devices, such as printing terminals, special word-processing printers (such as Diablo, Qume, or NEC Spinwriter mechanisms), or special output filter programs. These driving tables are written as C programs, compiled, and installed in **/usr/lib/term/tab***name*, where *name* is the name for that terminal type as shown in **term**(CT).

The structure of the tables is as follows. Sizes are in 240ths of an inch.

```
#define   INCH     240

struct termtable tlp ; { \* lp is the name of the term, *\
          int bset;    \* modify with new name, such as tnew *\
          int breset;
          int Hor;
          int Vert;
          int Newline;
          int Char;
          int Em;
          int Halfline;
          int Adj;
          char *twinit;
          char *twrest;
          char *twnl;
          char *hlr;
          char *hlf;
          char *flr;
          char *bdon;
          char *bdoff;
          char *iton;
          char *itoff;
          char *ploton;
          char *plotoff;
          char *up;
          char *down;
          char *right;
          char *left;
          char *codetab[256-32];
          char *zzz;
} ;
```

The meanings of the various fields are as follows:

bset bits to set in *termio.c_oflag* see **tty**(M) and **termio**(M)). after output.

breset bits to reset in *termio.c_oflag* before output.

Hor horizontal resolution in fractions of an inch.

Vert vertical resolution in fractions of an inch.

Newline space moved by a newline (linefeed) character in fractions of an inch.

Char quantum of character sizes, in fractions of an inch. (i.e., characters are multiples of Char units wide. See *codetab* below.)

Em size of an em in fractions of an inch.

Halfline space moved by a half-linefeed (or half-reverse-linefeed) character in fractions of an inch.

Adj quantum of white space for margin adjustment in the abscence of the **-e** option, in fractions of an inch. (i.e., white spaces are a multiple of Adj units wide)

Note: if this is less than the size of the space character (in units of Char; see below for how the sizes of characters are defined), *nroff* will output fractional spaces using plot mode. Also, if the **-e** switch to *nroff* is used, Adj is set equal to Hor by *nroff*.

twinit set of characters used to initialize the terminal in a mode suitable for *nroff*.

twrest set of characters used to restore the terminal to normal mode.

twnl set of characters used to move down one line.

hlr set of characters used to move up one-half line.

hlf set of characters used to move down one-half line.

flr set of characters used to move up one line.

bdon set of characters used to turn on hardware boldface mode, if any. *Nroff* assumes that boldface mode is reset automatically by the *twnl* string, because many letter-quality printers reset the boldface mode when they receive a carriage return; the *twnl* string should include whatever characters are necessary to reset the boldface mode.

bdoff set of characters used to turn off hardware boldface mode, if any.

iton set of characters used to turn on hardware italics mode, if any.

itoff set of characters used to turn off hardware italics mode, if any.

ploton set of characters used to turn on hardware plot mode (for Diablo-type mechanisms), if any.

plotoff set of characters used to turn off hardware plot mode (for Diablo-type mechanisms), if any.

up set of characters used to move up one resolution unit (Vert) in plot mode, if any.

down set of characters used to move down one resolution unit (Vert) in plot mode, if any.

right set of characters used to move right one resolution unit (Hor) in plot mode, if any.

left set of characters used to move left one resolution unit (Hor) in plot mode, if any.

codetab Array of sequences to print individual characters. Order is *nroff*'s internal ordering. See the file **/usr/lib/term/tabuser.c** for the exact order.

zzz a zero terminator at the end.

The *codetab* sequences each begin with a flag byte. The top bit indicates whether the sequence should be underlined in the **.ul** font. The rest of the byte is the width of the sequence in units of *Char*.

The remainder of each *codetab* sequence is a sequence of characters to be output. Characters with the top bit off are output as given; characters with the top bit on indicate escape into plot mode. When such an escape character is encountered, *nroff* shifts into plot mode, emitting *ploton*, and skips to the next character if the escape character was '\200'.

When in plot mode, characters with the top bit off are output as given. A character with the top bit on indicates a motion. The next bit indicates coordinate, with **1** being vertical and **0** being horizontal. The next bit indicates direction, with **1** meaning up or left.

The remaining five bits give the amount of the motion. An amount of zero causes exit from plot mode.

When plot mode is exited, either at the end of the string or via the amount-zero exit, *plotoff* is emitted followed by a blank.

All quantities which are in units of fractions of an inch should be expressed as INCH**num/denom*, where *num* and *denom* are respectively the numerator and denominator of the fraction; that is, 1/48 of an inch would be written as "INCH/48".

If any sequence of characters does not pertain to the output device, that sequence should be given as a null string.

The XENIX Development System must be installed on the computer to create a new driving table. The source code for a generic output device is in the file **/usr/lib/term/tabuser.c** Copy this file and make the necessary modifications, including the name of the termtable struct. Refer to the hardware manual for the codes needed for the output device (terminal, printer, etc.). Name the file according to the convention explained in **term**(CT). The makefile, **/usr/lib/term/makefile**, should be updated to include the source file to the new driving table. When the files are prepared, enter the command :

```
make cp
```

(See **make**(CP)). The source to the new driving table is linked with the object file **mkterm.o,** and the new driving table is created and installed in the proper directory.

FILES

/usr/lib/term/tab*name*	driving tables
/usr/lib/term/tabuser.c	generic source for driving tables
/usr/lib/term/makefile	makefile for creating driving tables
/usr/lib/term/mkterms.o	linkable object file for creating driving tables

SEE ALSO

nroff(CT), term(CT).

Notes

The XENIX Development System must be installed on the computer to create new driving tables.

Not all XENIX facilities support all of these options.

Name

terminfo – Format of compiled terminfo file.

Description

Compiled terminfo descriptions are placed under the directory **/usr/lib/terminfo**. In order to avoid a linear search of a huge XENIX system directory, a two-level scheme is used: **/usr/lib/terminfo/c/name** where *name* is the name of the terminal, and *c* is the first character of *name*. Thus, *act4* can be found in the file **/usr/lib/terminfo/a/act4**. Synonyms for the same terminal are implemented by multiple links to the same compiled file.

The format has been chosen so that it will be the same on all hardware. An 8- or more-bit byte is assumed, but no assumptions about byte ordering or sign extension are made.

The compiled file is created with the *tic*(C) program, and read by the routine *setupterm* in *terminfo*(S). The file is divided into six parts: the header, terminal names, boolean flags, numbers, strings, and string table.

The header section begins the file. This section contains six short integers in the format described below. These integers are (1) the magic number (octal 0432); (2) the size, in bytes, of the names section; (3) the number of bytes in the boolean section; (4) the number of short integers in the numbers section; (5) the number of offsets (short integers) in the strings section; (6) the size, in bytes, of the string table.

Short integers are stored in two 8-bit bytes. The first byte contains the least significant 8 bits of the value, and the second byte contains the most significant 8 bits. (Thus, the value represented is 256*second+first.) The value –1 is represented by 0377, 0377; other negative values are illegal. The –1 generally means that a capability is missing from this terminal. Note that this format corresponds to the hardware of the VAX and PDP-11. Machines in which this does not correspond to the hardware read the integers as two bytes and compute the result.

The terminal names section comes next. It contains the first line of the terminfo description, listing the various names for the terminal, separated by the '|' character. The section is terminated with an ASCII NUL character.

The boolean flags have one byte for each flag. This byte is either 0 or 1, as the flag is present or absent. The capabilities are in the same order as the file **<term.h>**.

Between the boolean section and the number section, a null byte will be inserted, if necessary, to ensure that the number section begins on an even byte. All short integers are aligned on a short-word boundary.

The numbers section is similar to the flags section. Each capability takes up two bytes, and is stored as a short integer. If the value represented is −1, the capability is taken to be missing.

The strings section is also similar. Each capability is stored as a short integer, in the format above. A value of −1 means the capability is missing. Otherwise, the value is taken as an offset from the beginning of the string table. Special characters in ^X or \c notation are stored in their interpreted form, not the printing representation. Padding information $<nn> and parameter information %x are stored intact in uninterpreted form.

The final section is the string table. It contains all the values of string capabilities referenced in the string section. Each string is null-terminated.

Note that it is possible for *setupterm* to expect a different set of capabilities than are actually present in the file. Either the database may have been updated since *setupterm* was recompiled (resulting in extra unrecognized entries in the file) or the program may have been recompiled more recently than the database was updated (resulting in missing entries). The routine *setupterm* must be prepared for both possibilities; this is why the numbers and sizes are included. Also, new capabilities must always be added at the end of the lists of boolean, number, and string capabilities.

As an example, an octal dump of the description for the Microterm ACT 4 is included:

```
microterm|act4|microterm act iv,
    cr=^M, cud1=^J, ind=^J, bel=^G, am, cub1=^H,
    ed=^_, el=^^, clear=^L, cup=^T%p1%c%p2%c,
    cols#80, lines#24, cuf1=^X, cuu1=^Z, home=^],

000 032 001      \0 025 \0 \b \0 212 \0  "  \0  m  i  c  r
020  o  t  e  r  m  |  a  c  t  4  |  m  i  c  r  o
040  t  e  r  m     a  c  t     i  v \0 \0 001 \0 \0
060 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
100 \0 \0  P \0 377 377 030 \0 377 377 377 377 377 377 377 377
120 377 377 377 377 \0 \0 002 \0 377 377 377 377 004 \0 006 \0
140 \b \0 377 377 377 377 \n \0 026 \0 030 \0 377 377 032 \0
160 377 377 377 377 034 \0 377 377 036 \0 377 377 377 377 377 377
200 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377
*
520 377 377 377 377     \0 377 377 377 377 377 377 377 377 377 377
540 377 377 377 377 377 377 007 \0 \r \0 \f \0 036 \0 037 \0
560 024  %  p  1  %  c  %  p  2  %  c \0 \n \0 035 \0
600 \b \0 030 \0 032 \0 \n \0
```

Some limitations: the total size of a compiled description cannot exceed 4096 bytes; the name field cannot exceed 128 bytes.

Files

/usr/lib/terminfo/*/*   compiled terminal capability data base

See Also

terminfo(M), terminfo(S), tic(C)

Name

types – Primitive system data types.

Syntax

#include <sys/types.h>

Description

The data types defined in the include file **<sys/types.h>** are used in XENIX system code; some data of these types are accessible to user code.

The form *daddr_t* is used for disk addresses except in an inode on disk, see *filesystem*(F). Times are encoded in seconds since 00:00:00 GMT, January 1, 1970. The major and minor parts of a device code specify kind and unit number of a device and are installation-dependent. Offsets are measured in bytes from the beginning of a file. The *label_t* variables are used to save the processor state while another process is running.

See Also

filesystem(F)

Replace this Page with Tab Marked:

Permuted Index

Permuted Index

Commands, System Calls, Library Routines and File Formats

This permuted index is derived from the "Name" description lines found on each reference manual page. Each *index* line shows the title of the entry to which the line refers, followed by the reference manual section letter where the page is found.

To use the *permuted index* search the middle column for a keyword or phrase. The right hand column contains the name and section letter of the manual page that documents the key word or phrase. The left column contains additional useful information about the command. Commands or routines are also listed in the context of the *index* line, followed by a colon (:). This denotes the "beginning" of the sentence. Notice that in many cases, the lines wrap, starting in the middle column and ending in the left column. A slash (/) indicates that the description line is truncated.

04-01-87

SCO-512-210-034