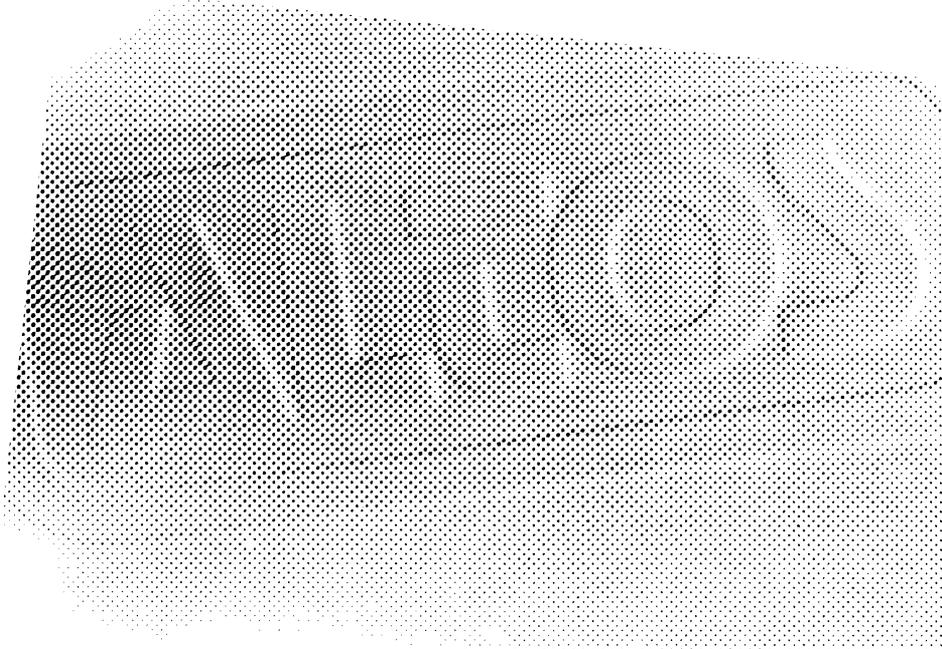




Altos UNIX[®] System V/386
Release 3.2

User's Guide



Altos UNIX[®] System V/386
Release 3.2

User's Guide

Document History

EDITION	PART NUMBER	DATE
Preliminary Edition	690-23408-001A	February 1990
First Edition	690-23408-001	April 1990
Second Edition	690-23408-002	March 1991

Copyright Notice

Manual Portions Copyright © 1990, 1991 Altos Computer Systems.

Manual Portions Copyright © 1989 AT&T.

Manual Portions Copyright © 1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988, 1989 Microsoft Corporation.

Manual Portions Copyright © 1983, 1984, 1985, 1986, 1987, 1988, 1989 The Santa Cruz Operation, Inc.

All rights reserved. Printed in U.S.A.

Unless you request and receive written permission from Altos Computer Systems, you may not copy any part of this document or the software you received, except in the normal use of the software or to make a backup copy of each diskette you received.

Trademarks

The Altos logo, as it appears in this manual, is a registered trademark of Altos Computer Systems.

386 and 486 are trademarks of Intel Corporation.

Microsoft, MS-DOS, and XENIX are registered trademarks of Microsoft Corporation.

UNIX is a registered trademark of UNIX System Laboratories, Inc.

Limitations

Altos Computer Systems reserves the right to make changes to the product described in this manual at any time and without notice. Neither Altos nor its suppliers make any warranty with respect to the accuracy of the information in this manual.

GUIDE TO YOUR ALTOS UNIX® SYSTEM V/386 RELEASE 3.2 DOCUMENTATION

RUN-TIME SYSTEM

These books come with every system:



Installation Guide

- Part Number: 690-24096-*nnn*
- Operating System installation Upgrade procedure



System Administrator's Guide

- Part Number: 690-23415-*nnn*
- Sysadmsh
 - Security
 - System tuning, troubleshooting
 - Peripherals
 - Virtual Disks



User's Guide

- Part Number: 690-23408-*nnn*
- Vi, ed, mail, awk, sed
 - Shells: sh and csh
 - Job scheduling commands



User's Reference (C, M, F)

- Part Number: 690-23414-*nnn*
(also provided online with each operating system)
- (C) Commands
 - (M) Miscellaneous files and commands
 - (F) File formats



System Administrator's Reference (ADM, HW)

- Part Number: 690-23416-*nnn*
(also provided online with each operating system)
- (ADM) Administrative commands
 - (HW) Hardware information

These books may be ordered separately:



Using the AOM Menu System

- Part Numbers: 690-23814-*nnn*
- Easy-to-use menus to use programs
 - Menu manager to add, update, remove menus



Tutorial

- Part Number: 690-23407-*nnn*
- Basic concepts and tasks
 - Files and directories
 - Utilities



International Operating System Guide

- Part Number: 690-23810-*nnn*
- Character sets
 - 7-bit vs. 8-bit characters

DEVELOPMENT SYSTEM

Set Part Number: 690-23417-000



Programmer's Reference (CP,S)

- (CP) Programming commands
- (S) System services, library routines



Programmer's Guide

- Lex, lint, yacc
- SCCS, make
- Extended Terminal Interface (ETI)
- Sdb, adb
- Shared libraries
- File and record locking



C Language Guide

- C User's Guide
- C Language Reference



Library Guide

- C Library Guide
- XENIX Development and Portability Guide
- International Development Guide



Developer's Guide

- DOS and OS/2 Development Guide
- STREAMS Primer
- STREAMS Programmer's Guide
- STREAMS Network Programmer's Guide



CodeView and Macro Assembler User's Guide

- The CodeView Debugger
- Macro Assembler User's Guide



Device Driver Writer's Guide

- Writing, compiling, and linking drivers
- SCSI drivers
- STREAMS and line disciplines
- (K) Kernel routines

To order any of the above manuals, call 408/434-6688, ext. 3004 and give the manual title and part number.

Operating System Documents for Different Audiences

As shown on the previous page, Altos offers many manuals with Altos UNIX System V—the manuals you receive will depend on your configuration. To help you decide which manuals are best suited to your needs, we have listed below the manuals according to three broad groups of users.

These lists are only suggested starting points in your search for information. They are not meant to imply that certain users should *not* read certain manuals. Find the user group that best applies to you, and use its list of manuals as a starting point for your reading, from which you can move on to other manuals.

Note that every Run-time System includes five manuals: the *Installation Guide*, the *User's Guide*, the *User's Reference*, the *System Administrator's Guide*, and the *System Administrator's Reference*. The Run-time System reference pages that describe the C, M, F, ADM, and HW commands ('man pages') are provided online as well. If you have the Development System, all manuals listed under "For Programmers:" come with your operating system. (All Development System reference pages are also provided online.) To order additional manuals, call (408) 434-6688, extension 3004 and give the manual title and part number.

For General Users (especially Beginners):

- Tutorial
- User's Guide
- User's Reference (C, M, F)
- Using the AOM Menu System

For System Administrators (and Advanced Users):

- Installation Guide
- System Administrator's Guide
- System Administrator's Reference (ADM, HW)
- International Operating System Guide
- Programmer's Reference (CP, S)

For Programmers:

- Programmer's Guide
- Programmer's Reference (CP, S)
- C Language Guide
- Library Guide
- Developer's Guide
- CodeView and Macro Assembler User's Guide
- Device Driver Writer's Guide

Contents

1 Introduction

- Overview 1-1
- About This Guide 1-2
- Notational Conventions 1-4

2 vi: A Text Editor

- Introduction 2-1
- Demonstration 2-2
- Editing Tasks 2-18
- Solving Common Problems 2-55
- Setting Up Your Environment 2-57
- Summary of Commands 2-64

3 ed

- Introduction 3-1
- Demonstration 3-2
- Basic Concepts 3-3
- Tasks 3-4
- Context and Regular Expressions 3-33
- Speeding Up Editing 3-50
- Cutting and Pasting with the editor 3-55
- Editing Scripts 3-57
- Summary of Commands 3-58

4 mail

- Introduction 4-1
- Basic Concepts 4-2
- Using mail 4-9
- Leaving Compose Mode Temporarily 4-18
- Setting Up Your Environment 4-21
- Using Advanced Features 4-24

5 Communicating with Other Sites

- Introduction 5-1
- Using Micnet 5-2
- Using UUCP 5-6
- Logging in to Remote Systems 5-15

6 bc: A Calculator

- Introduction 6-1
- Demonstration 6-2
- Tasks 6-5
- Language Reference 6-16

7 The Shell

- Introduction 7-1
- Basic Concepts 7-2
- Shell Variables 7-11
- The Shell State 7-18
- A Command's Environment 7-20
- Invoking the Shell 7-22
- Passing Arguments to Shell Procedures 7-23
- Controlling the Flow of Control 7-26
- Special Shell Commands 7-40
- Creation and Organization of Shell Procedures 7-44
- More About Execution Flags 7-46
- Supporting Commands and Features 7-47
- Effective and Efficient Shell Programming 7-55
- Shell Procedure Examples 7-60
- Shell Grammar 7-68

8 The C-Shell

- Introduction 8-1
- Invoking the C-shell 8-2
- Using Shell Variables 8-4
- Using the C-Shell History List 8-7
- Using Aliases 8-10
- Redirecting Input and Output 8-12
- Creating Background and Foreground Jobs 8-13
- Using Built-In Commands 8-14
- Creating Command Scripts 8-17
- Using the argv Variable 8-18
- Substituting Shell Variables 8-19
- Using Expressions 8-21
- Using the C-Shell: A Sample Script 8-22
- Using Other Control Structures 8-25
- Supplying Input to Commands 8-26
- Catching Interrupts 8-27
- Using Other Features 8-28
- Starting a Loop at a Terminal 8-29
- Using Braces with Arguments 8-31
- Substituting Commands 8-32
- Special Characters 8-33

9 The Korn Shell

- Introduction 9-1
- Starting ksh 9-2
- Using the ksh Built-in Editors 9-3
- Accessing Commands in the History File 9-8
- Using Job Control 9-10
- Customizing the ksh Environment 9-14
- Manipulating Commands Wider Than the Screen 9-19
- Using Expanded cd Capabilities 9-20

10 Using A Trusted System

- Introduction 10-1
- Login Security 10-3
- Using Commands On A Trusted System 10-7
- Recommended Security Practices 10-12
- Data Encryption—Commands and Descriptions 10-16

11 Simple Programming with awk

- Introduction 11-1
- Basic awk 11-2
- Patterns 11-11
- Actions 11-18
- Output 11-34
- Input 11-39
- Using awk with Other Commands and the Shell 11-45
- Example Applications 11-48
- awk Summary 11-53

12 Using the Stream Editor: sed

- Introduction 12-1
- Overall Operation 12-2
- Addresses 12-4
- Functions 12-6

13 Using the Job Scheduling Commands: at, cron and batch

- Introduction 13-1
- Automatic Program Execution with cron 13-2
- Delaying Program Execution with batch and at 13-4

14 Using DOS Accessing Utilities

Introduction 14-1

Accessing DOS Files with the dos(C) Utilities 14-2

Using Mounted DOS Filesystems 14-5

Index

Change Information

Chapter 1

Introduction

Overview 1-1

About This Guide 1-2

Notational Conventions 1-4



Overview

1

This guide provides extensive information on several of the most useful Altos UNIX System V facilities, including **mail**, the **vi** and **ed** text editors, **uucp**, **micnet** and **bc**, the UNIX “desktop calculator.” In addition, the guide includes information on the three UNIX “shells”: the Bourne shell, the C shell, and the new Korn shell.

Note

The last section of this manual, “Change Information,” summarizes the changes that have been made to the manual since the previous version.

About This Guide

This guide is organized as follows:

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

Chapter 2, “vi: A Text Editor” explains how to use the UNIX fullscreen editor, **vi**.

Chapter 3, “ed” explains how to use the UNIX line editor, **ed**.

Chapter 4, “mail” explains how to use the UNIX electronic mail facility.

Chapter 5, “Communicating with Other Sites” explains how to transfer files to and from and how to execute commands on other computer sites. These other sites might be XENIX or UNIX sites, but they do not need to be. They can, for instance, be MS-DOS™ sites.

Chapter 6, “bc: A Calculator” explains how to use **bc**, a sophisticated calculator program.

Chapter 7, “The Shell” explains how to use the powerful features of the UNIX Bourne shell.

Chapter 8, “The C-Shell” explains how to use the powerful features of the UNIX C shell.

Chapter 9, “The Korn Shell” explains how to use the new enhancement available with the UNIX Korn shell (**ksh**).

Chapter 10, “Using A Trusted System” discusses the security features that may be in use at your site and how to work with them.

Chapter 11, “Simple Programming with awk” shows how to write simple programs that can be used to manipulate files and data.

Chapter 12, “Using the Stream Editor: sed” demonstrates automated file editing.

Chapter 13, “Using the Job Scheduling Commands: cron, at, and batch” demonstrates how to schedule or delay the execution of programs and utilities.

Chapter 14, “Using the DOS Accessing Utilities” explains how to access DOS files indirectly using the DOS utilities, or directly using mounted DOS filesystems.

Notational Conventions

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

Initial Capitals

Initial Capitals indicate the name of a command or mode. When a command is introduced it is followed by the keystroke that invokes it, (i.e., the Insert (i) command).

boldface

Boldface indicates a command, option, flag, or program name to be entered as shown. Keystrokes are boldfaced when they indicate a command to enter as shown, (i.e., enter the **i** command and press **<Return>**). Commands that are issued while within a program, such as a file editor like **vi(C)**, are not boldfaced so they will not be confused with commands given to the shell.

Boldface indicates the name of a UNIX utility or library routine. (To find more information on a given utility, consult the “Alphabetized List” in the appropriate *Reference* for the manual page that describes it.)

italics

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

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

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

Italics indicate a reference to part of an example.

Italics indicate emphasized words or phrases in text.

screen font

This font is used for screen displays and messages.

[]

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

Brackets indicate the position of the cursor in text examples.

...

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

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

“ ”

Quotation marks indicate the first use of a technical term.

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

Chapter 2

vi: A Text Editor

Introduction 2-1

Demonstration 2-2

- Entering the Editor 2-2
- Inserting Text 2-3
- Repeating a Command 2-4
- Undoing a Command 2-4
- Moving the Cursor 2-5
- Deleting 2-6
- Searching for a Pattern 2-10
- Searching and Replacing 2-11
- Leaving vi 2-13
- Adding Text From Another File 2-13
- Leaving vi Temporarily 2-14
- Changing Your Display 2-15
- Canceling an Editing Session 2-16

Editing Tasks 2-18

- How to Enter the Editor 2-18
- Moving the Cursor 2-19
- Moving Around in a File: Scrolling 2-22
- Inserting Text Before the Cursor: i and I 2-23
- Appending After the Cursor: a and A 2-24
- Correcting Typing Mistakes 2-25
- Opening a New Line 2-25
- Repeating the Last Insertion 2-25
- Inserting Text From Other Files 2-25
- Inserting Control Characters into Text 2-30
- Joining and Breaking Lines 2-30
- Deleting a Character: x and X 2-30
- Deleting a Word: dw 2-31
- Deleting a Line: D and dd 2-31
- Deleting an Entire Insertion 2-32
- Deleting and Replacing Text 2-32
- Moving Text 2-36
- Searching: / and ? 2-40
- Searching and Replacing 2-41

- Pattern Matching 2-43
- Undoing a Command: u 2-46
- Repeating a Command: . 2-47
- Leaving the Editor 2-48
- Editing a Series of Files 2-49
- Editing a New File Without Leaving the Editor 2-51
- Leaving the Editor Temporarily: Shell Escapes 2-52
- Performing a Series of Line-Oriented Commands: Q 2-53
- Finding Out What File You're In 2-54
- Finding Out What Line You're On 2-54

Solving Common Problems 2-55

Setting Up Your Environment 2-57

- Setting the Terminal Type 2-57
- Setting Options: The set Command 2-58
- Displaying Tabs and End-of-Line: list 2-59
- Ignoring Case in Search Commands: ignorecase 2-59
- Displaying Line Numbers: number 2-59
- Printing the Number of Lines Changed: report 2-60
- Changing the Terminal Type:term 2-60
- Shortening Error Messages: terse 2-60
- Turning Off Warnings: warn 2-61
- Permitting Special Characters in Searches: nomagic 2-61
- Limiting Searches: wrapscan 2-61
- Turning on Messages: msg 2-61
- Mapping Keys 2-62
- Abbreviating Strings 2-62
- Customizing Your Environment: The .exrc File 2-63

Summary of Commands 2-64

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 Altos UNIX System V, **ed** and **vi**. **ed** is discussed in the “ed” chapter of this manual.

2

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 *User's Reference* instead of this chapter.

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

Demonstration

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

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

Entering the Editor

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

```
vi temp
```

Your screen will look like this:

```

-
-
-
-
-
-
-
-
-
-
-
-
"temp" [New file]
```

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

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

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

Inserting Text

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

i

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

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

```
-
-
-
-
-
```

Press the **<ESC>** key when you are finished.

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

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

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

Demonstration

Repeating a Command

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

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

Undoing a Command

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

u

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

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

```
-
-
-
-
-
```

2

Now enter:

```
u
```

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

Moving the Cursor

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

```
h Left
l Right
k Up
j Down
```

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

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

Demonstration

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

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

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

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

Deleting

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

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

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

`1G`

At first, your file should look like this:

2

```
[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`

Demonstration

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:

```

contains_
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

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.

Demonstration

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 <CTL>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%--
```

2

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

Searching and Replacing

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

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

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

Demonstration

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

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 UNIX 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 UNIX prompt appears.

Adding Text From Another File

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

```
vi practice
```

Demonstration

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

```
:r temp
```

Your file should look like this:

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

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

Leaving vi Temporarily

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

```
!:date
```

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

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



Changing Your Display

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

To turn on automatic line numbering, enter:

```
:set number
```

Demonstration

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

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

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

```
:set all
```

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

Canceling an Editing Session

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

```
:q!
```

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

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

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

Editing Tasks

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

How to Enter the Editor

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

With a Filename

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

```
vi filename
```

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

At a Particular Line

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

```
vi +100 filename
```

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

At a Particular Word

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

```
vi +/word filename
```

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

2

```
vi +/contain temp
```

Moving the Cursor

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

Moving the Cursor by Characters: **h**, **l**, **f**, **F**, **t**, **T**, **<Space>**, **<BKSP>**

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

```
4h
```

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

```
Fp
```

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

```
fp
```

Editing Tasks

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

Tp

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

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

Moving the Cursor by Lines: **j**, **k**

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

3j

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

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

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

No, I didn't know he had returned.

If you press:

6w

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

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

6W

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

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

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

4b

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

Enter:

4B

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

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

Moving the Cursor by Lines

Forward: **j**, **<CTL>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 **<CTL>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

Editing Tasks

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**, **<CTL>p**

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

4k

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

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

Moving Around in a File: Scrolling

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

Scrolling Up Part of the Screen: **<CTL>u**

<CTL>u scrolls up one-half screen.

Scrolling Up the Full Screen: <CTL>b

<CTL>b scrolls up a full screen.

Scrolling Down Part of the Screen: <CTL>d

<CTL>d scrolls down one-half screen.

2**Scrolling Down a Full Screen: <CTL>f**

<CTL>f scrolls down a full screen.

Placing a Line at the Top of the Screen: z

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

z

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

33z

Press <Return>, and line 33 scrolls to the top of the screen. For information on how to display line numbers, see the section “Displaying Line Numbers: number.”

Inserting Text Before the Cursor: i and I

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

i

Editing Tasks

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

Anywhere on a Line: *i*

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

At the Beginning of the Line: **I**

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

Appending After the Cursor: *a* and **A**

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

Anywhere on a Line: *a*

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

At the end of a Line: **A**

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

Correcting Typing Mistakes

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

Opening a New Line

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

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

Repeating the Last Insertion

`<CTL>@` repeats the last insertion. Press `i` to enter Insert mode, then press `<CTL>@`.

`<CTL>@` only repeats insertions of 128 characters or less. If more than 128 characters were inserted, `<CTL>@` does nothing.

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

Inserting Text From Other Files

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

```
:r filename
```

Editing Tasks

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

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

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

2. Enter:

```
:e filename
```

where *filename* is the file that contains the text you want to copy, and press `<Return>`.

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

4. Enter:

```
mk
```

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

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

```
"ay'k
```

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

6. Enter:

```
:e#
```

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

```
"ap
```

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

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

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

2

Copying Lines From Elsewhere in the File

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

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

```
:nu
```

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

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

Editing Tasks

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

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

The result is:

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

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

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

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

```
"ayy
```

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

2. Enter the following line:

```
"ap
```

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

Your screen looks like this:

2

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

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

```
"a3yy
```

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

```
2yw
```

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

```
10yl
```

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

```
y$
```

Inserting Control Characters into Text

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

Joining and Breaking Lines

To join two lines press:

J

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

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

r

then press <Return>.

Deleting a Character: x and X

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

3x

To delete three characters preceding the cursor, enter:

3X

Deleting a Word: dw

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

```
3dw
```

Deleting a Line: D and dd

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

```
3dd
```

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

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

```
:200,250d
```

Press **<Return>**.

When the command finishes, the message:

```
50 lines
```

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

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

```
:$d
```

Editing Tasks

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 *User's Reference*.

Deleting an Entire Insertion

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

Deleting and Replacing Text

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

Overstriking: **r** and **R**

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

```
rb
```

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

```
4rb
```

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

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

2

```
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.
~
~
~
~
~
```

Editing Tasks

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`

Editing Tasks

Press `<Return>`. This line-oriented command means “Substitute (s) for the word *removing* the word *deleting*, everywhere it occurs on the current line (g)”. If you don’t include a `g` at the end, only the first occurrence of *removing* is changed.

For more information on using line-oriented commands to replace text, see the section “Searching and Replacing.”

Moving Text

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

```
set number
```

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

The following example uses the `number` option. For other ways to display line numbers, see the section “Finding Out What Line You’re On.”

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

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

```
:2,3m4
```

Your screen should look like this:

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

2

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

Editing Tasks

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:

2

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

Now enter:

"2p

The line from the *first* deletion appears:

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

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

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

Editing Tasks

Searching: / and ?

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

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

```
/account
```

Press <Return>. The cursor is placed on the first character of the pattern. To place the cursor at the beginning of the line above “account”, for example, enter:

```
/account/-
```

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

```
/account/-2
```

To place the cursor two lines below “account”, enter:

```
/account/+2
```

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

```
?account
```

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

```
/U\.S\.A\./
```

You can continue to search for a pattern by pressing:

n

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

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

```
:set ignorecase
```

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

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

```
:set nowrapscan
```

and press <Return> to prevent searches from wrapping. For more information about setting options, see the section “Setting Up Your Environment.”

Searching and Replacing

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

The syntax of a search and replace command is:

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

Editing Tasks

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 **<CTL>r**.

2

Choosing a Replacement

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

To run this command on the example file, enter:

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

Press **<Return>**. The first instance of “contain” appears on the status line:

```
Files contain text.
```

Press **y**, then **<Return>**. The next occurrence of *contain* appears.

Pattern Matching

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

Editing Tasks

Matching the Beginning of a Line

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

```
^text/
```

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

Matching the End of a Line

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

```
text$/
```

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

Matching Any Single Character

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

```
/.ed /
```

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

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

Matching a Range of Characters

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

```
[a-z]
```

finds any lowercase letter. The search pattern:

```
[aA]pple/
```

finds all occurrences of “apple” and “Apple”.

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

Matching Exceptions

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

```
[^a-z]
```

finds anything but a lowercase letter or a newline.

Matching the Special Characters

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

```
^/
```

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

```
:set nomagic
```

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

Editing Tasks

To restore *magic*, enter:

```
:set magic
```

For more information about setting options, see the “Setting Up Your Environment” section.

Undoing a Command: **u**

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

Use the following line as an example:

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

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

```
Text [1]ines.  
-  
-  
-  
-  
-  
-  
-  
-
```

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

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

2

If you press **u** again, “contains” is deleted again:

```
Text [l]ines.
-
-
-
-
-
-
-
-
-
```

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

Repeating a Command: .

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

```
2dw
```

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

Editing Tasks

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

Leaving the Editor

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

```
:x
```

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

```
ZZ
```

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

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

```
:q!
```

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

```
:q
```

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

```
No write since last change (:quit! overrides)
```

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

Saving a File Without Leaving the Editor

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

```
:w
```

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

```
:w filename
```

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

Editing a Series of Files

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

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

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

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

```
vi file*
```

Editing Tasks

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

```
vi *.s
```

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

```
vi [3-5]*.s
```

or

```
vi [a-h]*
```

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

```
vi ??????.*
```

For more information on using special characters, see “Naming Conventions” in the “Basic Concepts” chapter of the *Tutorial*.

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

```
x files to edit
```

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

```
:n
```

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

If you forget what files you are editing, enter:

```
:args
```

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

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

```
:e file4
```

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

```
:n
```

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

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

```
:rew
```

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

```
:rew!
```

Editing a New File Without Leaving the Editor

You can start editing another file anywhere on a UNIX 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*.

Editing Tasks

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/joel/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/joel/memo*.

Leaving the Editor Temporarily: Shell Escapes

You can execute any UNIX 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 UNIX command. To send mail to joe without leaving the editor, enter:

```
!:mail joe
```

Type your message and send it. (For more information about the UNIX mail system, see the “mail” chapter.) After you send it, the message

```
[Press return to continue]
```

appears. Press <Return> to continue editing.

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

```
:!sh
```

The UNIX prompt appears. You may execute as many commands as you like. Press `<CTL>d` to terminate the new shell and return to your file.

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

```
[No write since last change]
```

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

```
:set nowarn
```

For more information about setting options in `vi`, see the section “Setting Up Your Environment.”

Performing a Series of Line-Oriented Commands:

Q

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

```
Q
```

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

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

```
vi
```

Editing Tasks

Finding Out What File You're In

If you forget what file you are editing, press `<CTL>g` while you are in Command mode. A line similar to the following appears on the status line:

```
``memo'' [Modified] line 12 of 100 --12%--
```

From left to right, the following information is displayed:

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

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

The same information can be obtained by entering:

```
:file
```

or

```
:f
```

Finding Out What Line You're On

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

```
:nu
```

and press `<Return>`. This command displays the current line number and the text of the line.

To display line numbers for the entire file, see the section “Displaying Line Numbers: number.”

Solving Common Problems

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

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

Press `<ESC>` until the bell rings. When the bell rings you are in Command mode.

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

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

```
[Press return to continue]
```

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

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

- *There are extra characters on my screen.*

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

Solving Common Problems

- *The system crashed while I was editing.*

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

```
vi -r filename
```

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

- *I keep getting a colon on the status line when I press <Return>*

You are in line-oriented Command mode. Enter:

```
vi
```

to return to normal vi Command mode.

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

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

```
:wq
```

and press <Return>. Turn to the section “Setting the Terminal Type” for information on how to set your terminal type correctly.

Setting Up Your Environment

There are a number of options that can be set that affect your terminal type, how files and error messages are displayed on your screen, and how searches are performed. These options can be set with the `set` command while you are editing, they can be defined with the `EXINIT` environment variable (see the *environ(M)* manual page), or they can be placed in the `vi .exrc` startup file (see “Customizing Your Environment: The `.exrc` File”).

You can also define mappings and abbreviations to reduce repetitive tasks with the `map` and `abbr` commands while you are editing, with `EXINIT`, or in the `.exrc` file.

The following sections describe how to set some commonly used options and how to create mappings and abbreviations. There is a complete list of options in `vi(C)` in the *User's Reference*.

Setting the Terminal Type

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

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

Setting the `TERM` Variable: The Bourne Shell

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

```
TERM=2621
export TERM
```

Setting Up Your Environment

Setting the TERM Variable: The C Shell

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

```
setenv TERM 2621
```

Setting Options: The set Command

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

Listing the Available Options

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

```
:set all
```

Your display should look similar to this:

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

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

Setting an Option

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

```
set noignorecase
```

Displaying Tabs and End-of-Line: list

2

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 `<CTL>i` (`^I`) represents the tab character.

Ignoring Case in Search Commands: ignorecase

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

```
:set ignorecase
```

To change this option, enter:

```
:set noignorecase
```

Displaying Line Numbers: number

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

```
:set number
```

This redraws your screen. Numbers appear to the left of the text.

Setting Up Your Environment

Printing the Number of Lines Changed: report

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

```
:set report=1
```

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

```
report=5
```

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

Changing the Terminal Type:term

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

```
:set term
```

Press **<Return>**. See the section “Setting the Terminal Type” for more information about TERM variables.

Shortening Error Messages: terse

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

```
:set terse
```

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

```
No write since last change, quit! overrides
```

becomes:

```
No write
```

Turning Off Warnings: warn

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

```
:set nowarn
```

Permitting Special Characters in Searches: nomagic

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

```
:set nomagic
```

Limiting Searches: wrapscan

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

```
:set nowrapscan
```

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

Turning on Messages: mesg

If someone sends you a message with the **write** command while you are in **vi** the text of the message will appear on your screen. To remove the message from your display you must press <CTL>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
```

Setting Up Your Environment

Mapping Keys

The **map** command maps any character or escape sequence to a command sequence. For example, with the following command defined, when you enter the pound sign (#) in Command mode, vi adds a semicolon to the end of the current line.

```
map # A;^[
```

⟨CTL⟩[represents the ESC key you must enter to exit from Insert mode. When you create a mapping, use ⟨CTL⟩v to escape control characters.

Here is a more complex example:

```
map ^P :w^M:!spell %^M
```

⟨CTL⟩p key is mapped to two commands; it writes the file, then executes a shell escape to run the spell checker on the current file (represented by the percent sign). The ⟨CTL⟩m represents the ⟨Return⟩ you must enter to execute each command.

Be careful not to map keys that are already defined within vi, such as ⟨CTL⟩r, which is defined by default to redraw the screen.

You can remove a mapping with the **unmap** command.

Abbreviating Strings

The **abbr** command allows you to avoid typing a frequently used word or phrase by mapping a short string to a longer string. For example, with the following command defined, when you enter “Usa” in Insert mode, vi expands the string to “United States of America”.

```
:abbr Usa United States of America
```

When you create an abbreviation, it helps to use mixed case (as in “Usa”) so that you can still enter “USA” if you need to without it expanding.

You can remove an abbreviation with the **unabbreviate** command.

Customizing Your Environment: The .exrc File

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

```
set number
set ignorecase
set nowarn
set report=1
map ^W !}fmt^M
abbr unix \s-lUNIX\s+1
```

Each time you invoke `vi` with the above settings, your file is displayed with line numbers, case is ignored in searches, warnings before shell escape commands are turned off, and any command that modifies more than one line will display a message indicating how many lines were changed. In addition, the `<CTL>w` key is defined to escape to the shell to run a formatting command on the current paragraph, and the string “unix” is defined to expand to a string containing `troff(CT)` commands that print small capital letters.

Summary of Commands

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

Entering vi

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

Cursor Movement

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

Summary of Commands

Inserting Text

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

Delete Commands

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

Change Commands

Command	Function
<code>cw</code>	Changes 1 word
<code>3cw</code>	Changes 3 words
<code>cc</code>	Changes current line
<code>5cc</code>	Changes 5 lines

2

Search Commands

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

Summary of Commands

Search and Replace Commands

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

Pattern Matching: Special Characters

This character:	Matches:
<code>^</code>	Beginning of a line
<code>\$</code>	End of a line
<code>.</code>	Any single character
<code>[]</code>	A range of characters

Leaving vi

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

Summary of Commands

Options

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

Chapter 3

ed

Introduction 3-1

Demonstration 3-2

Basic Concepts 3-3

 The Editing Buffer 3-3

 Commands 3-3

 Line Numbers 3-3

Tasks 3-4

 Entering and Exiting The Editor 3-4

 Appending Text: a 3-5

 Writing Out a File: w 3-6

 Leaving The Editor: q 3-7

 Editing a New File: e 3-8

 Changing the File to Write Out to: f 3-8

 Reading in a File: r 3-9

 Displaying Lines On The Screen: p 3-10

 Displaying the Current Line: dot (.) 3-13

 Deleting Lines: d 3-15

 Performing Text Substitutions: s 3-16

 Searching 3-19

 Changing and Inserting Text: c and i 3-23

 Moving Lines: m 3-25

 Performing Global Commands: g and v 3-26

 Displaying Tabs and Control Characters: l 3-29

 Undoing Commands: u 3-30

 Marking Your Spot in a File: k 3-30

 Transferring Lines: t 3-31

 Escaping to the Shell: ! 3-32

Context and Regular Expressions 3-33

 Period: (.) 3-34

 Backslash: \ 3-36

 Dollar Sign: \$ 3-39

 Caret: ^ 3-41

 Star: * 3-41

Brackets: [and] 3-44
Ampersand: & 3-45
Substituting New Lines 3-47
Joining Lines 3-48
Rearranging a Line: \ (and \) 3-48

Speeding Up Editing 3-50
Semicolon: ; 3-52
Interrupting the editor 3-54

Cutting and Pasting with the editor 3-55
Inserting One File Into Another 3-55
Writing Out Part of a File 3-55

Editing Scripts 3-57

Summary of Commands 3-58

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` is very similar to `ed`, and therefore this chapter can be used as an introduction to `ex` as well as to `ed`.

Demonstration

This section leads you through a simple session with **ed**, giving you a feel for how it is used and how it works. To begin the demonstration, invoke **ed** by entering:

```
ed
```

This invokes the editor and begins your editing session. **ed** has no prompt unless `-p 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
```

Follow this with `<CTL>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.

Basic Concepts

This section illustrates some of the basic concepts that you need to understand to effectively use `ed`.

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.

3

Commands

Commands are entered at your keyboard. Like normal UNIX commands, entry of a command is ended by entering a `<Return>`. After you enter `<Return>` the command is carried out. In the following examples, we will presume that entry of each command is completed by entering a `<Return>`, 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.

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.

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.

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.

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 `<CTL>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.

Tasks

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**, and if you have configured **ed** to provide details of the error, it will tell you by displaying the message:

```
?
error message
```

For an explanation of how to turn the error message display on or off, refer to “Commands” in the User’s Reference. By default, the error message display is turned off.

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 add-

ing 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.

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 UNIX 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
```

Tasks

Next, use the **cat** command to display the file on your terminal screen to see that everything has worked.

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.

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*.

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

Tasks

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
```

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 `` 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:

Tasks

`$-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`

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.

Tasks

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.

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:

```
Or filename
```

Tasks

and you can enter lines at the beginning of the buffer by entering:

```
Oa
[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.

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/something else/p
```

which makes a correction on the current line, then prints it to make sure the correction worked out right. If it didn’t, you can try again. (Notice that the **p** is on the same line as the **s** command. With few exceptions, **p** can follow any command; no other multicommand lines are legal.)

Tasks

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  
.br/>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.

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.
```

3

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
```

Tasks

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 the section “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/
```

Tasks

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.

3

If you get unexpected results with any of the characters

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

read the section “Context and Regular Expressions.”

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

Tasks

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.

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
$r temp
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-linemafter-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
```

Tasks

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 the section “Marking Your Spot in a File: `k`” in this chapter.

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/dump/p
```

prints all lines that contain the word “dump.” 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 the section “Context and Regular Expressions” in this chapter.

3

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 “DUMP” to “dump” everywhere, and verify that it really worked, with:

```
g/DUMP/s//dump/gp
```

Notice that we used // in the substitute command to mean “the previous pattern” in this case, “dump.” The **p** command executes on each line that matches the pattern, not just on those in which a substitution took place.

Tasks

The global command makes two passes over the file. On the first pass, all lines that match the pattern are marked. On the second pass, each marked line is examined in turn, dot is set to that line, and the command executed. This means that it is possible for the command that follows a `g` or `v` command to use addresses, set dot, and so on, quite freely. For example:

```
g/^\.P/+
```

prints the line that follows each “.P” command (the signal for a new paragraph in some formatting packages). Remember that plus (+) means “one line past dot.” And:

```
g/topic/? \.H?p
```

searches for each line that contains the word “topic” scans backwards until it finds a line that begins with a “.H” (a heading) and prints it, thus showing the headings under which “topic” is mentioned. Finally:

```
g/^\.EQ+ ,/\.EN/-p
```

prints all the lines that lie between lines beginning with “.EQ” and “.EN” formatting commands.

The `g` and `v` commands can also be preceded by line numbers, in which case the lines searched are only those in the range specified.

It is possible to give more than one command under the control of a global command. For example, suppose the task is to change “x” to “y” and “a” to “b” on all lines that contain “thing.” Then:

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

is sufficient. The backslash (\) signals the `g` command that the set of commands continues on the next line; the `g` command terminates on the first line that does not end with a backslash.

Note that you cannot use a substitute command to insert a new line within a `g` command. 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/\
.nf\
.sp
```

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

Displaying Tabs and Control Characters: l

ed provides two commands for printing the contents of the text you are editing. You should already be familiar with **p**, in combinations like:

```
l,$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.

Tasks

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.

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
```

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.

3

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 (*italics* are comments):

```
a
Now is the time for all good men to come to the aid of their party.
.
t.                               [make a copy]
s/men/women/                     [change it a bit]
t.                               [make third copy]
s/Now is/yesterday was/          [change it a bit]
```

Your file will look like this:

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

Escaping to the Shell: !

Sometimes it is convenient to temporarily escape from the editor to execute a UNIX 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 UNIX 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.

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 UNIX commands and utilities, including `grep(C)`, `sed(C)` (See the *User's Reference*).

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.

Context and Regular Expressions

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.

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/./,/
```

Context and Regular Expressions

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.

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 any-

thing” 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 to be a single real period.

The backslash can also be used when searching for lines that contain a special character. Suppose you are looking for a line that contains:

```
.DE
```

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:

```
/\\
```

Context and Regular Expressions

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/
```

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,
```

Context and Regular Expressions

The command needed is:

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

to get:

```
Now is the time, for all good men.
```

The dollar sign (\$), here, provides context to make specific which comma we mean. Without it, the s command would operate on the first comma to produce:

```
Now is the time. for all good men,
```

To convert:

```
Now is the time.
```

into:

```
Now is the time?
```

as we did earlier, we can use:

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

Like the period (.), the dollar sign (\$) has multiple meanings depending on context. In the following line:

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

the first "\$" refers to the last line of the file, the second refers to the end of that line, and the third is a literal dollar sign to be added to that line.

Caret: ^

The caret (^) stands for the beginning of the line. For example, suppose you are looking for a line that begins with “the.” If you 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$/
```

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:

Context and Regular Expressions

```
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-----y text
```

then all minus signs (-) can be replaced by a single space with the command:

```
s/x-*y/x y/
```

Finally, suppose that the line was:

```
text x.....y text
```

If you enter:

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

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

For example, if the line is:

```
x text x.....y text y
```

then entering:

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

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/x y/
```

Now the substitution works, for “\.*” means “as many periods as possible.”

There are times when the pattern “.” is exactly what you want. For example, to change:

```
Now is the time for all good men ....
```

into:

```
Now is the time.
```

use “.” to remove everything after the “for.”

```
s/ for.*./
```

3

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
```

Context and Regular Expressions

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.

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.

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
```

Context and Regular Expressions

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.

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.

Context and Regular Expressions

When a new line is substituted in a string, dot is left at the last line created.

Joining Lines

Lines may be joined together, with the `j` command. Assume that you are given the lines:

```
Now is  
the time
```

Suppose that dot is set to the first line. Then the command:

```
j
```

joins them together to produce:

```
Now is the time
```

No 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.

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.

3

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.

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:

3

```
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:

Or

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.

Speeding Up Editing

(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.

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?;??
```

Speeding Up Editing

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.

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.

Cutting and Pasting with the editor

This section describes how to manipulate pieces of files, individual lines or groups of lines.

Inserting One File Into Another

Suppose you have a file called *memo*, and you want the file called *table* to be inserted just after a reference to Table 1. That is, in *memo*, somewhere is a line that reads:

3

```
Table 1 shows that ...
```

and the data contained in *table* has to go there.

To put *table* into the correct place in the file edit *memo*, find “Table 1” and add the file *table* right there:

```
ed memo
/ Table 1/
response from ed
.r table
```

The critical line is the last one. The `r` command reads a file; here you asked for it to be read in right after line dot. An `r` command, without any address, adds lines at the end, so it is the same as “`$.r`.”

Writing Out Part of a File

The other side of the coin is writing out part of the document you’re editing. For example, you may want to split the table from the previous example 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
```

Cutting and Pasting with the editor

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/w table
```

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
.
```

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 ‘DUMP’ to ‘dump’ and every ‘USA’ to ‘America’ in a large number of files. Enter the following lines into the file *script*:

```
g/DUMP/s//dump/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 UNIX shell command interpreter, you can cycle through a set of files automatically. The dash (-) suppresses unwanted messages from **ed**.

When preparing editing scripts, you may need to place a period as the only character on a line to indicate termination of input from an **a** or **i** command. This is difficult to do in **ed**, because the period you type will terminate input rather than be inserted in the file. Using a backslash to escape the period won’t work either. One solution is to create the script using a character such as the at-sign (@), to indicate end of input. Then, later, use the following command to replace the at-sign with a period:

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

Summary of Commands

The following is a list of all **ed** commands. The general form of **ed** commands is the command name, preceded by one or two optional line numbers and, in the case of **e**, **f**, **r**, and **w**, followed by a filename. Only one command is allowed per line, but a **p** command may follow any other command (except **e**, **f**, **r**, **w**, and **q**).

Command	Description
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 <i>g/string/commands</i> executes <i>commands</i> on those lines that contain <i>string</i> , which can be any context search expression.
i	Inserts lines before specified line (or dot) until a single period is typed on a new line. Dot is set to the last line inserted.

- l** Lists lines, making visible nonprinting ASCII characters and tabs. Otherwise similar to **p**.
- m** Moves lines specified to after the line named after **m**. Dot is set to the last line moved.
- p** Prints specified lines. If none are specified, print the line specified by dot. A single line number is equivalent to a *line-number* **p** command. A single **<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 do not contain *string*.
- v** The command *v/string/commands* executes *commands* on those lines that do not contain *string*.
- u** Undoes the last substitute command.
- w** Writes out the editing buffer to a file. Dot remains unchanged.
- =** Prints value of dot. (An equal sign by itself prints the value of \$.)

Summary of Commands

- !command* The line *!command* causes *command* to be executed as a UNIX 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 *\$*.

Chapter 4

mail

Introduction 4-1

Basic Concepts 4-2

Mailboxes 4-2

Messages 4-3

Modes 4-3

Headers 4-5

Command Syntax 4-7

Message Lists 4-7

Using mail 4-9

Composing and Sending a Message 4-9

Reading Your Mail 4-10

Saving a Message 4-12

Replying to a Message 4-13

Deleting a Message 4-14

Forwarding Mail: f and F 4-14

Executing Shell Commands 4-15

Sending Mail to Remote Sites 4-15

Leaving mail: q and x 4-17

Leaving Compose Mode Temporarily 4-18

Editing Headers: ~t, ~c, ~b, ~s, ~R, and ~h 4-18

Adding a File to the Message: ~r and ~d 4-19

Enclosing Another Message: ~m and ~M 4-20

Setting Up Your Environment 4-21

Creating Mailing Lists: a 4-22

Keeping Mail in the System Mailbox: hold 4-23

The Cc Prompt: askcc 4-23

Listing Messages in Chronological Order 4-23

Using Advanced Features 4-24

Using mail as a Reminder Service 4-24

Handling Large Amounts of Mail 4-25

Introduction

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

This chapter is organized to satisfy the needs of both beginning and advanced users. The first section discusses basic **mail** concepts. The second section provides demonstrations of the most commonly used commands. Later sections describe the more advanced **mail** commands and some advanced uses of **mail**.

For a quick introduction to get started using **mail** immediately, see the *Tutorial*. For a complete list of **mail** functions, refer to **mail(C)** in the *User's Reference*.

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
- message lists
- headers
- command syntax

Mailboxes

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

1. The user mailbox is *not* the place where mail is initially routed—that place is the system mailbox in the directory */usr/spool/mail*.
2. Mail is not picked up *from* your user mailbox.

Messages

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

- To: This field is mandatory. It contains one or more valid user names to which you can 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 is optional.
- 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 is optional.

4

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

Modes

The **mail** program provides two distinct functions: sending mail and managing messages. **mail**'s two main modes are *compose mode* and *command mode*. You create a message in compose mode. In command mode, you perform **mail** operations for managing your mail.

The most common way of using **mail** is to begin a session by entering:

mail

If you have mail waiting, this command automatically places you in command mode. In this mode, you can enter commands for handling your mail. If you have no mail waiting, you see the message “No mail in /usr/spool/mail/login” and are returned to the UNIX shell.

Basic Concepts

From the shell or from **mail** command mode, you can enter compose mode to create a message with:

mail user

where *user* is the user name of the person to whom you want to send mail. In compose mode, you can enter the text of your message ending each line with a `<Return>`. Send the message by pressing `<CTL>d` on a new line; you then exit from the **mail** program and return to the UNIX shell. From compose mode, you can issue commands called *compose escapes* that allow you to temporarily leave or escape from compose mode. Compose escapes, which must be entered at the beginning of a line, begin with a tilde (`~`) and so are also called *tilde escapes*.

Once you have pressed `<Return>` to end a line of a message you are creating, you cannot change that line from within compose mode. You must enter *edit mode* to change the line. In edit mode, you edit the body of a message using the full capabilities of an editor.

To enter edit mode from compose mode, use the compose escape `~e` to enter `ed`, the line editor, or `~v` to enter `vi`, the visual editor. (`vi` might not be available on your system.) It is often useful to be able to invoke either a line or visual editor, depending on the type of terminal you are using. When you finish editing the message, write it out and quit the editor; **mail** responds with:

(continue)

You are now back in compose mode and can continue creating your message.

You can also enter edit mode from command mode to edit any existing message. Use either the **edit** (`e`) or the **visual** (`v`) command. When you write out the message and quit the editor, **mail** reads the message back into the message buffer.

If you want to mail a message that already exists in a file, you can do so from the command line (without entering `mail`) as follows:

```
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.

When invoking `mail` from the shell, certain `mail` command-line options are available. Two useful command-line options are the `-s` “subject” option and the `-c` “carbon copy” option. You can specify a subject and carbon copy recipients on the command line with these options. For example, you could send a file named *note* with the subject line “Important Meeting” by entering the following command:

```
mail -s "Important Meeting" -c "ted bob" bill joe sue < note
```

The *To:* field will contain *bill*, *joe*, and *sue*; the *Cc:* field will contain *ted* and *bob*.

All command-line options must appear before the list of users for the *To:* field. If an argument to an option contains multiple words, the entire argument set must be enclosed in quotes. Other command-line options are described in the `mail(C)` manual page.

Headers

When you enter `mail` command mode, a list of message headers is displayed that looks something like this:

```

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

```

Basic Concepts

By default, **mail** displays headers in reverse chronological order, the most recent message is displayed at the top of the list. The messages are numbered in ascending order from first received to most recently received; the message at the top of the list has the highest number. You can change the order in which headers are displayed by setting the **chron** and **mchron** options.

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 header contains:

- a greater-than sign (>) pointing to the current message
- a status indicator: "N" for new and "U" for unread
- the number of the message
- the sender
- the date sent
- the number of lines and characters
- the subject (if the message contains a *Subject:* field)

Message headers are displayed a screenful at a time. You can set the size of a screen with the **screen=** option. You can move forward one screenful with the **headers (h)** command:

h +

You can move backward one screenful with **h -**. Both plus and minus take an optional numeric argument that indicates the number of header windows to move forward or backward before printing. With no argument at all, the **headers** command displays a window of headers in which the header of the current message is at the center.

The following are some characteristics of the header list:

- Deleted messages do not appear in the listing.
- Messages saved with the **save** or **write** command are flagged with a star (*).
- Messages selected with the **mbox** command to be saved in your user mailbox are flagged with an "M".
- Messages held in the system mailbox with the **hold** or **preserve** command are flagged with a "P".

Command Syntax

Each **mail** command has its own syntax. Some commands take no arguments, some take only one, and others take several arguments.

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; you can use its unique abbreviation. For example, you can enter “s” instead of “save” for the **save** command “se” instead of “set” for the **set** command. Throughout this chapter, the appropriate abbreviation is enclosed in parentheses after the name of the 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.

Message Lists

Many **mail** commands take a list of messages as an argument. A *message list* is a list of message identifiers, ranges, users, search strings, or message types separated by spaces or tabs. For commands that take a message list as an argument, if no message list is given, the current message is used.

Message identifiers can be either decimal numbers, which directly specify messages, or one of three special characters: ^ (caret), . (dot), and \$ (dollar sign), which specify the first, current, and last *non-deleted* message, respectively.

A range of messages is two message identifiers separated by a dash. To display the headers of all the messages from the current message to the last message, enter:

```
h .-$
```

By giving a user name as part of a message list, you can display the messages sent by a particular user. For example, if you want to read only the messages sent by your manager, enter:

```
p markt
```

The **print** (**p**) command displays those messages on the screen one after another.

Basic Concepts

You can use a search string to specify all messages with the given string in the subject line (case is ignored). For example, to display the headers of only the messages with “meeting” in the subject line, enter:

```
h /meeting
```

You can create a message list by defining the type of messages in which you are interested. Use a colon followed by one of the following key letters:

```
d   deleted messages
n   new messages
o   old messages
r   read messages
u   unread messages
```

For example, to see a list of headers of the messages you deleted, enter:

```
h :d
```

Message lists can contain combinations of numbers, ranges, and names. For example, to delete all messages about your print jobs from user *lp* that are numbered from the first non-deleted message to 7 or 11 and 12, use the **delete** (**d**) command with the following message list:

```
d lp ^7 11 12
```

As a shorthand notation, you can specify an asterisk (*) to mean all non-deleted messages. For example, to completely clean out your mailbox, use the **save** (**s**) command with an asterisk and a filename to save all non-deleted messages to the specified file.

```
s * mail.old
```

The asterisk symbol cannot be used with any other message list notation.

Using mail

This section demonstrates some of **mail**'s more commonly used features. Refer to the **mail(C)** manual page for details about other commands.

Composing and Sending a Message

Try sending a message to yourself by entering the following command from UNIX command level:

```
mail self
```

where *self* is your user name.

If the **asksub** option is set, **mail** prompts for a subject line.

```
Subject: Sample Message
```

Enter a one-line summary of the message, then press **<Return>** to enter compose mode.

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 **<CTL>u** to kill a line and **<BKSP>** to back up one character.

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 <CTL>d on a new line to end the message.
```

To view the message you are composing (including the heading fields) as it will appear when you send it, enter:

```
~p
```

Using mail

This will display the following:

```
-----  
Message contains:  
To: self  
Subject: Sample Message  
  
This is a message sent to myself.  
I compose a message by entering lines of text.  
Press Ctrl-d on a new line to end the message.  
(continue)
```

You can abort a message you are composing by entering two interrupts in a row (i.e., pressing INTERRUPT twice), and the message is not sent. When you abort a message, a copy of the body of the undelivered message is saved in the file *dead.letter* in your home directory.

When you are ready to send your message, press <CTL>d on a line by itself to end the message and to send it. Once you have sent mail, there is no way to undo the act, so be careful.

If mail cannot be delivered to the address you specified, you will be notified via return mail, which will include the undeliverable message.

Reading Your Mail

The message you sent yourself should have arrived in your system mailbox. To begin a mail session, enter:

```
mail
```

mail then displays a sign-on message and a list of message headers:

```
Altos UNIX System V Mail (Version n.n) Type ? for help.  
"/usr/spool/mail/self": 1 message  
> N 1 self Fri Aug 31 12:26 9/229 "Sample Message"  
&
```

The ampersand prompts you to enter a **mail** command. You can set the prompt to a different string with the **prompt=** option. To get help on all the available **mail** commands, enter:

?

Next, to display the message that you sent to yourself, press **<Return>**. **mail** displays:

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

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

4

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. If a subject or a carbon copy (Cc:) field was specified by the sender, they too are displayed when you read the message.

You can configure your environment so that you are notified whenever new mail is sent to you even if you are not in **mail**. To do so, you should set the **MAIL** shell variable if you are using the Bourne shell or the **mail** shell variable if you are using the C-shell. For more information, see “The Shell” chapter of the *User’s Guide* and **csH(C)** in the *User’s Reference*.

After examining a message, you will most likely want to either leave the message in your system mailbox, save it in a file, reply to it, or delete it. These and other useful **mail** operations are described in the next sections.

Saving a Message

Sometimes you might want to save a message for future reference. If you leave **mail** with the **quit** (**q**) command without performing any other operation on your message, the message is normally saved in the user mailbox. **mail** displays the following message before returning you to the UNIX shell.

```
Saved 1 message in /u/self/mbox
```

To keep the message in the system mailbox, use the **hold** or **preserve** command. **mail** displays the following message.

```
Held 1 message in /usr/spool/mail/self
```

You will see the same message next time you invoke **mail**.

Saving many messages in the user or system mailbox can be confusing and can slow down processing. You can use the **save** (**s**) command to organize your mail by putting messages that relate to each other in a specific file. The **save** command writes out each message to the file given as the last argument on the command line. For example, the following command appends the current message to the file *letters* :

```
s letters
```

The file *letters* is created if it does not already exist. **Save** writes out the entire message, including the *To:*, *Subject:*, and *Cc:* fields. **mail** now treats the file *letters* as a mail folder.

Each saved message is marked with an asterisk (*). When you quit from **mail**, saved messages are normally deleted from the system mailbox.

You can access messages saved in a mail folder by specifying the filename with the **mail -f** command-line option or with the **folder (fold)** or **file (fi)** command from within **mail**. Both of these methods read in the specified file, giving you access to the messages in that folder in the same way you have access to the messages in your system mailbox when you invoke **mail** normally. Your user mailbox is also a mail folder; its messages can be accessed in the same way.

Note

If you leave a mail folder by switching to another folder or back to your system or user mailbox, you can no longer get back a deleted message from the original folder. As far as the **undelete** command is concerned, leaving a mail folder is like exiting from **mail**.

Another way to save a message is with the **lpr (l)** command, which sends the message to the lineprinter. This command takes a message list as its argument, then paginates and prints each message on the lineprinter. For example:

```
l doug
```

prints each message from the user *doug*.

Replying to a Message

Often, you want to deal with a message by responding to its author right away. You can use the **reply (r)** command to set up a response to a message, automatically addressing a reply to the person who sent the original 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, and messages are terminated by pressing **(CTL)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.

Deleting a Message

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

```
d1
```

prevents **mail** from retaining message 1 in the user mailbox. The message disappears altogether, along with its number.

The **dp** command deletes the current message and displays the next message, which is useful for quickly reading and disposing of mail.

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 1, enter:

```
u1
```

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

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. 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 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.

Executing Shell Commands

You can execute a shell command without leaving **mail** from either **mail** command mode or compose mode. From command mode, precede the command with an exclamation point. For example:

```
!date
```

displays the current date without leaving **mail**.

From compose mode, precede the command with **^!**. The command is executed, and you are returned to **mail** compose mode without altering your message.

From command mode, you can enter a new shell with the **shell (sh)** command. To exit from this new shell and return to **mail** command mode, press **<CTL>d**.

4

Sending Mail to Remote Sites

You can send mail to users on remote computer sites that are networked to your own site. The network can either be a Micnet network or a UUCP network. Ask your system administrator if you are not sure which network the site you want to mail to uses.

If the site you want to send mail to is a Micnet site, you would enter the following command to mail to a user on that site:

```
mail user@site-name
```

Note that the user name is followed by an at symbol (**@**).

For example, to send mail to *stevem* on the Micnet computer named *obie*, you would enter the following command:

```
mail stevem@obie
```

After entering this command, you would continue with **mail** just as if you were sending mail to a local user.

Using mail

You can also send mail to users on remote UUCP sites. To find out which UUCP sites your computer communicates with, enter the following command at the UNIX prompt:

```
uname
```

A list of site names is displayed.

To send mail to a user on a UUCP site, enter the following command:

```
mail site-name!user
```

The site name must be followed by an exclamation point (!).

For example, to send mail to user *markt* on site *bowie*, you would enter the following command:

```
mail bowie!markt
```

You would then proceed to use **mail** just as if you were mailing to a local user.

You can enter several site names on a command line; be sure to follow each one with an exclamation point. As another example, suppose your site talked to UUCP site *bowie* and that *bowie* talked to UUCP site *bradley*. You could send mail to user *cindy* on *bradley* by entering the following command:

```
mail bowie!bradley!cindy
```

Note

If you are using the C-shell, you must “escape” exclamation points with the backslash (\). A C-shell user would enter the above command as follows:

```
mail bowie\!bradley\!cindy
```

For more information on communicating with remote sites, see the “Communicating with Other Sites” chapter in this guide.

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 user mailbox, except the following:

- deleted messages, which are discarded irretrievably
- messages marked with the **hold** or **preserve** command, which are saved in your system mailbox; if the **hold** option is set, messages that you have read are automatically saved in your system mailbox
- messages saved with the **save** or **write** command

Forwarded messages are *not* removed from the system mailbox.

If you want 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 in your user mailbox.

Leaving Compose Mode Temporarily

While composing a message to be sent to others, you might need to change heading fields, invoke the text editor on a partial message, execute a shell command, or perform some other useful function. **mail** provides these capabilities through *compose escapes*, which consist of a tilde (~) at the beginning of a line, followed by a one- or two-character command that specifies the function to be performed.

To get a list of the available compose escapes, enter the following command from compose mode:

```
~?
```

The **mail(C)** manual page contains details about these compose escapes, which are available *only* when you are composing a new message; they have no meaning when you are in **mail** command mode.

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

```
~c name1 name2 ...
```

adds the named people to the *Cc:* list. Similarly, the escape:

```
~b name1 name2 ...
```

adds the named people to the *Bcc:* (Blind carbon copy) list. The people on this list receive a copy of the message, but are not mentioned anywhere in the message you send.

The recipients of the message are given in the *To:* field; the subject is given in the *Subject:* field, and carbon copy recipients are given in the *Cc:* field. If you want to edit these in ways impossible with the `~t`, `~s`, and `~c` 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 UNIX 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, and another brings you to the *Bcc:* 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. Remember that you can always use `~p` to see what the message looks like.

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.

Leaving Compose Mode Temporarily

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** normally copies the text of your message buffer to *dead.letter* whenever you abort the creation of a message. You can abort the message by entering two consecutive interrupts or by entering a `~q` escape.

Enclosing Another Message: `~m` and `~M`

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

`~m 4`

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

`~M 4`

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

Setting Up Your Environment

You can define your **mail** environment with switch and string options that 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 can be specified on a line. For example, you might have a **set** command that looks like this:

```
set dot askcc SHELL=/usr/bin/sh
```

The options **dot** and **askcc** are switch options; **SHELL** is a string option.

The **set** command with no arguments displays the options currently set.

You can create a personal mailing list with the **alias** (**a**) command. By using an alias, you can send mail to one name and have it go to a group of people. With no arguments, **alias** displays all currently defined aliases. With one argument, it displays the users defined by the given alias.

It is most useful to place **set**, **unset**, and **alias** commands in the file *.mailrc* in your home directory, where they define your personal default environment when you invoke **mail**. 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 **set** options and system-wide aliases

are defined in */usr/lib/mail/mailrc*. These are installed by whoever is in charge of your system. Personal aliases and personal **set** options are defined in *.mailrc*.

Setting Up Your Environment

The following is a sample *.mailrc* file:

```
# number sign introduces comments
# personal aliases office and cohorts are defined below
alias office bill joe sue
alias cohorts john mary bob beth mike

# set dot lets messages be terminated by period on new line
# set asksub prompts for Subject: before entering compose mode

set dot asksub

# changes to always begin executing from the same directory

cd
```

The following sections demonstrate how to create mailing lists and describe a few of the common *set* options. Refer to the *mail(C)* manual page for details about other options.

Creating Mailing Lists: a

The *alias* 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*.

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 *beatles* will read:

```
To: john paul george ringo
```

and not:

```
To: beatles
```

Keeping Mail in the System Mailbox: **hold**

The **hold** option determines whether messages remain in the system mailbox when you exit **mail**. If you do not set **hold**, 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.

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 edit your message.

Listing Messages in Chronological Order

The *chron* switch causes messages to be listed and displayed in chronological order. By default, messages are listed and displayed 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 want to list the headers of the most recently received mail first but read the messages themselves in chronological order.

Using Advanced Features

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

Using mail as a Reminder Service

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

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

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

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

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

For information about writing shell procedures, see "The Shell" chapter in this guide.

Handling Large Amounts of Mail

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

When in doubt, throw it out.

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

The previously mentioned measures are not always helpful enough in organizing the many messages that you are likely to receive. Another effective approach is to save mail in files organized by sender, by topic, or by a combination of the two. However, be forewarned—this approach to organizing mail quickly eats up disk space. Using mail folders is described in “Saving a Message”.

4

You can create a directory to hold your mail folders and define that directory to **mail** with the **folder=** option. Then, whenever you save a message without giving a pathname, **mail** puts the message in a file (or folder) in that directory. For example, if you want to save your messages by default in the directory *mail* in your home directory, use:

```
set folder=mail
```

If you forget the names of your mail folders, you can use the **folders** command to display the names of the files in the directory set by the **folder=** option.

Chapter 5

Communicating with Other Sites

Introduction 5-1

Using Micnet 5-2

 Transferring Files with rcp 5-2

 Executing Commands with remote 5-4

 Transferring Files with mail 5-5

Using UUCP 5-6

 Transferring Files with uucp 5-6

 Transferring Files with uuto 5-11

 Executing Commands with uux 5-13

Logging in to Remote Systems 5-15

 Using ct 5-15

 Using cu 5-17

Introduction

UNIX systems include a series of utilities that allow you to communicate with other computer sites. The particular utilities you use depend on how your computer is connected to the other site, what tasks you want to accomplish on the other site, and what operating system is running on the other site.

If the site is in close proximity to your computer, in the same room, for example, then it is likely that the two computers are connected by a simple serial line. If the site is a UNIX site, use the Micnet commands discussed in “Using Micnet” below to transfer files between the two sites and to execute commands on the remote site. If the site is a UNIX site, use the UUCP commands discussed in “Using UUCP” below.

If, on the other hand, the site you want to communicate with is on another floor, or across the country, your computer is connected to it by telephone lines. If the site is a UNIX or XENIX site, use the UUCP commands discussed in “Using UUCP” below to transfer files between the two sites and execute commands on the remote site. If the site is not a UNIX or XENIX site, use the commands discussed in “Using cu” below.

Neither the UUCP commands nor the Micnet commands allow you to have an *interactive* session with the remote site. If you want to have an interactive session, use the commands discussed in “Using cu” below.

This chapter assumes that your UUCP and/or Micnet networks are configured already. If this is not true, refer to “Building a Remote Network with UUCP” and “Building a Local Network with Micnet” in the *System Administrator's Guide* for more information.

Using Micnet

A Micnet network is a network of two or more computers connected by serial communication lines. A serial communication line is a cable with RS-232 connectors on each end.

The computers in a Micnet network use three commands to “talk” to one another. These are **rcp**, **remote** and **mail**. The **rcp** command is used to transfer files between machines in the network. The **remote** command is used to execute UNIX commands on a remote Micnet machine. The **mail** command is used to communicate with users on a remote computer. Each of these commands is discussed in the following sections.

Transferring Files with rcp

The **rcp** command is used to transfer copies of both text and binary files between machines connected in a Micnet network. Its syntax is similar to that of the **cp** command:

```
rcp [options] [src_computer:]src_file [dest_computer:]dest_file
```

These arguments mean the following:

src_file	The name of the file that you want to copy.
src_computer	The name of the computer on which <i>src_file</i> is located.
dest_file	The name of the copied file on the receiving computer. Usually, <i>src_file</i> and <i>dest_file</i> are the same.
dest_computer	The name of the computer on which <i>dest_file</i> is located.

You must have read permission on the source file and read and execute permissions on the directory that contains the source file in order to copy it with **rcp**. In addition, you must have write permission on the directory on the computer that is to receive the source file.

As an example, suppose you have three computers named *machine1*, *machine2* and *machine3* connected in a Micnet network. Suppose also that you want to send a copy of a file named *transfile* in the */usr/markt*

directory on *machine1* to the */tmp* directory on *machine3*. To do so, enter the following command:

```
rcp machine1:/usr/markt/transfile machine3:/tmp/transfile
```

If you are in the directory that contains the source file, specify the filename only. You do not have to specify the full machine and path-name. Using the example above, enter the following command from */usr/markt* on *machine1* to copy *transfile* to */tmp* on *machine3*:

```
rcp transfile machine3:/tmp/transfile
```

In addition to using **rcp** to send copies of files to remote computers, you can use **rcp** to retrieve copies of files from remote computers. Using the example above, suppose that *machine3* is your local computer and that you want to get a copy of */usr/markt/transfile* from *machine1*. To do so, enter the following command:

```
rcp machine1:/usr/markt/transfile /tmp/transfile
```

This command would place a copy of */usr/markt/transfile* on *machine1* in the */tmp* directory on *machine3*.

5

Because files are not sent immediately, an **rcp** transfer may take a few minutes. Files are copied to a spool directory and sent when the appropriate daemons “awaken.” (A daemon is a program that periodically runs in the background.) In the case of **rcp**, the daemon that transfers files is the **daemon.mn** daemon.

rcp Options

Two options are available for use with **rcp**. These are **-m** and **-u [machine:]user**. The **-m** option causes mail to be sent to the user who entered the **rcp** command, reporting on the success or failure of the transfer. If you want **mail** to report to another user, use **-u [machine:]user**. This causes **mail** to report to *user* on *machine*.

The following command, issued from */usr/markt* on *machine1*, sends a copy of */usr/markt/transfile* on *machine1* to the */tmp* directory on *machine3*. Since the **-m** option is specified, mail will be sent reporting on the success or failure of the command:

```
rcp -m transfile machine3:/tmp/transfile
```

Using Micnet

For more information on the **rcp** command, see **rcp(C)**.

Executing Commands with remote

The **remote** command allows execution of commands across serial lines. The syntax of the **remote** command is:

```
remote [options] site_name command [arguments]
```

If the remote command produces output, that output is mailed to your system mailbox. Otherwise, **remote** sends mail only if the remote command fails to execute.

As an example, suppose that you are working on *machine1* and that you want to list the contents of the */tmp* directory on *machine2*. To do so, enter the following command:

```
remote machine2 ls /tmp
```

Since the **ls** command produces output, the output is mailed to you. In this case, your mail contains a listing of the contents of */tmp* on *machine2*.

remote Options

Two very useful options to the **remote** command are the **-m** and **-f file** options. The **-m** option sends mail to you reporting on the success or failure of the command execution. Suppose, for example, that you want to remove */test* from */tmp/markt* on *machine2*. To do so, enter the following command:

```
remote -m machine2 rm /tmp/markt/test
```

After this command is executed, you receive mail reporting on the success or failure of the **rm** command.

The **-f file** option allows you to specify a file on the local computer that contains the input for the command that is to be executed on the remote computer. As an example, suppose that you have a file named *chapter1* on your local computer that you want to print on *machine2*'s default printer. To do so, enter the following command:

```
remote -m -f chapter1 machine2 lp
```

Because the **-m** option is specified, you are informed by mail of the success or failure of the **remote** command.

Note

The system administrator can specify which commands are allowed to execute remotely over serial lines on which computers. The commands that are allowed to execute remotely on a UNIX system are listed in the computer's */etc/default/micnet* file. Any UNIX command can execute remotely if the computer's */etc/default/micnet* file contains the statement *executeall* on a line by itself.

Transferring Files with mail

The **mail** command can be used to transfer files between computers in a Micnet network. However, there are several drawbacks to using **mail** for this purpose:

- You must transfer the file to a *user* on the remote system, rather than to a *directory*.
- You can only use **mail** to transfer small files. Large files are randomly truncated by **mail**.
- You cannot transfer binary files with **mail**.

On the other hand, **mail** is very useful for sending small files to several users at once on a remote system. For information on using **mail**, see “mail” in this guide.

Using UUCP

UUCP is a series of programs that provide networking capabilities for UNIX systems. While UUCP commands can be used over serial lines, they are usually used on computers connected by telephone lines.

The UUCP programs allow you to transfer files between remote computers and to execute commands on remote computers. Since the computers may be connected by telephone lines, UUCP transfers can take place over thousands of miles. A UUCP site in New York City can transfer a file to or execute a command on a connected UUCP site in San Francisco, or Jakarta, or anywhere in the world. The following sections explain how to use these UUCP programs.

Transferring Files with uucp

Both the **uucp** and **uuto** commands can be used to transfer copies of binary and text files between remote UUCP sites. There are advantages and disadvantages to each. The **uucp** command gives you great flexibility in specifying where on the remote system the transferred file is to be placed. However, **uucp** syntax can be rather long and complicated. The **uuto** command, on the other hand, is easy to use. But **uuto** restricts where you can place the file on the remote system. In addition, retrieving a file sent with **uuto** is slightly more complicated than retrieving a file sent with **uucp**.

The **uucp** command is discussed in this section. The **uuto** command is discussed in the following section.

Before You Begin

Before you can copy files to remote sites with **uucp**, you must verify that:

- Your local site is a “dial out” site.
- Your local site “knows” how to call the remote site.
- The files that you want to send have read permission set for others.
- The directory that contains the file that you want to send has read and execute permissions set for others.

- Your computer has write permission in the directory on the remote site to which you want to copy the file.

Each of these is discussed below.

Some UUCP sites are “dial-in” sites, some are “dial-out” sites, and some are both. Verify that your site is a dial-out site. If it is not, your computer might have the capability to be on the receiving end of a UUCP connection, but not on the calling end.

You must be sure that your computer “talks” to the site with which you want to communicate. The `uname` command gives you this information. Entering `uname` with no options lists the UUCP sites your computer talks to directly. Entering `uname` with the `-l` option causes the name of your computer to be displayed.

Note that you may be able to communicate with a site that does not show up in a `uname` listing. This is possible because UUCP sites are often “chained together.” So if you know that a site you want to transfer files to communicates with a site that your system communicates with, you can send files to the first site through the second. An example is provided below under “Indirect Transfers.”

In order to copy a file to a remote UUCP site, the file must have read permission set for others and the directory that contains the file must have read and execute permissions set for others. Use the `l` command to examine the file’s permissions and the `l -d` command to examine the directory’s permissions. If the permissions are not correct, enter the following commands to set the correct permissions:

```
chmod o+r filename
chmod o+rx directory
```

Finally, you must verify that your computer has write permission on the directory on the remote site to which you want to transfer files. Each remote UUCP site has a `/usr/lib/uucp/Permissions` file. This file specifies the directories on that site from which your computer can read and to which your computer can write. You can only send a file to a directory on a remote site if your computer has write permissions on that directory, as specified on the remote site’s `/usr/lib/uucp/Permissions` file.

By default, most UUCP sites permit calling-in computers to write to their `/usr/spool/uucppublic` directory. Since there is no way to find out which directories your computer can write to on the remote site, short of contacting somebody at the site, the safest thing to do when making a UUCP transfer is to write to `/usr/spool/uucppublic`. The procedure for doing this is outlined below.

Using UUCP

Using uucp

The syntax of the **uucp** command is similar to the syntax of **cp**:

```
uucp [options] src_computer!src_file dest_computer!dest_file
```

These arguments mean the following:

src_file	The name of the file that you want to copy.
src_computer	The name of the computer on which <i>src_file</i> is located.
dest_file	The name of the copied file on the receiving computer. Usually, <i>src_file</i> and <i>dest_file</i> are the same.
dest_computer	The name of the computer on which <i>dest_file</i> is located.

There are several different ways to specify the location on the remote machine to which you want to transfer the file. The simplest is the *~/dest_file* specification. This is also the safest specification, because *~/dest_file* is expanded to */usr/spool/uucppublic/dest_file*, thereby assuring that the transfer will succeed.

For example, to send */usr/markt/transfile* on *machine1* to */usr/spool/uucppublic* on *machine2*, enter the following command:

```
uucp /usr/markt/transfile machine2!~/transfile
```

This command creates the file */usr/spool/uucppublic/transfile* on *machine2*.

If */usr/markt* is your current directory, you can copy *transfile* to *machine2* with the following command:

```
uucp transfile machine2!~/transfile
```

The **uucp** command works much like the **rcp** command. Files are not copied and sent immediately. Instead, copies are placed in a spool directory and sent once the appropriate daemon awakens. In the case of the UUCP programs, the daemon is the **uucico** daemon. Depending on how your system is configured, a **uucp** transfer might take place within minutes, or it might take hours.

Note

Since the exclamation mark has special meaning to the C-shell, you must “escape” with a backslash (\) any exclamation marks that appear in a **uucp** command, if you are using the C-shell. For a C-shell user, the command above is specified as:

```
uucp transfile machine2!\~/transfile
```

Another form of the command allows you to specify the full pathname of the copied file on the remote computer. This is for sending the file to a specific directory on the remote system. However, you must be sure that your computer has write permission on this directory, otherwise the transfer will fail.

As an example, suppose that you want to send *transfile* in */usr/markt* on *machine1* to the */usr/cindy* directory *machine2*. To do so, enter the following command:

```
uucp /usr/markt/transfile machine2!/usr/cindy/transfile
```

5

Note that, like the **rcp** command, the **uucp** command can be used to retrieve files from a remote site, in addition to copying files to a remote site. Using the example above, if your local computer is *machine2* and you want to send a copy of */usr/markt/transfile* on *machine1* to the */usr/cindy* directory on *machine2*, enter the following command:

```
uucp machine1!/usr/markt/transfile /usr/cindy/transfile
```

You can also use *~user* to specify a location on the remote computer. The *~user* argument is expanded to the pathname of the home directory of the person on the remote computer whose login is *user*. For example, if */usr/cindy* is the home directory of a user whose login is *cindy* on *machine2*, enter the following command from the */usr/markt* directory on *machine1* to copy */usr/markt/transfile* to */usr/cindy*:

```
uucp transfile machine2!~cindy/transfile
```

The receiving computer expands *~cindy* to the full pathname of *cindy*'s home directory, creating */usr/cindy/transfile*. Again, your computer must have write permission in *cindy*'s home directory in order for this transfer to succeed.

Using UUCP

Indirect Transfers

You might be able to send files to a UUCP site not listed in a `uname` listing. As an example, suppose that your local computer is connected to a UUCP site named *machine2*. Suppose also that *machine2* is connected to a UUCP site named *machine3*. You can send `/tmp/transfile` on your local computer to `/usr/spool/uucppublic` on *machine3*. Do so by specifying the full UUCP address relative to your local computer:

```
uucp /tmp/transfile machine2!machine3!~/transfile
```

Note that each site name in the command line is followed by an exclamation mark. By placing several site names in a `uucp` command line, you can greatly extend the range of systems to which you can copy files with `uucp`. This is also true for the `uuto` and `uux` commands discussed below.

uucp Options

Several options are available for the `uucp` command. Some of the most useful are the `-m` and `-n user` options.

The `-m` option sends you mail reporting on the success or failure of the file transfer. The `-n user` option notifies the *user* on the machine to whom the files are sent of the file transfer.

Other options are available for use with `uucp`. Refer to `uucp(C)` for a complete list of these options.

Checking the Status with uustat

You can use the `uustat` command to check on the status of files you copied with `uucp`. To check on the status of all your `uucp` jobs, enter the following command:

```
uustat
```

Your output looks like the following:

```
1234 markt machine2 2/19-10:29 2/19-10:40 JOB IS QUEUED
```

Reading from left to right, the elements of this message are:

1234	This is the job number assigned to this uucp transfer.
markt	This is the user who requested the transfer.
machine2	This is the site name of the recipient's computer.
2/19-10:29	This is the date and time the job was queued in the spool directory.
2/19-10:40	This is the date and time of the uustat request.
<i>Job Status</i>	This message tells you the status of the job. In this case, JOB IS QUEUED tells you that the job is in the spool directory waiting to be sent. When the transfer is completed, uustat displays the message: COPY FINISHED, JOB DELETED

Several options are available for use with **uustat**. Refer to **uustat(C)** for more information.

5

Transferring Files with **uuto**

The **uuto** command allows you to copy files to the public directory of a UUCP site to which your system is connected. The public directory on most UNIX and XENIX systems is */usr/spool/uucppublic*. The syntax of **uuto** is:

```
uuto [options] source_file destination_computer!login
```

The *login* argument is the login of the user to whom you are sending files.

Before you can send a file with **uuto**, you must verify that:

- The file has read permission set for others.
- The directory that contains the file has read and execute permissions set for others.

Using UUCP

If the permissions are not correct, enter the following commands to set the correct permissions:

```
chmod o+r filename  
chmod o+rx directory
```

Files sent with **uuto** are placed in the directory:

```
/usr/spool/uucppublic/receive/login/source_computer
```

In this example, *login* is the login of the user to whom you are sending files and *source_computer* is the site name of *your* system.

As an example, suppose that you want to send a copy of *transfile* in */tmp* on your computer, *machine1*, to a user whose login is *cindy* on *machine2*. To do so, enter the following command:

```
uuto /tmp/transfile machine2!cindy
```

This command copies *transfile* to the following directory:

```
usr/spool/uucppublic/receive/cindy/machine1
```

When the file transfer is complete, the recipient is notified by **mail** that the file has arrived. If the **-m** option is used on the **uuto** command line, the sender is notified by **mail** of the success or failure of the transfer.

Like **uucp**, files transferred with **uuto** are not transferred immediately after the command is entered. Instead, they are placed in a spool directory and sent when the **uucico** daemon awakens.

Retrieving Files with **uupick**

In order to retrieve a file sent by **uuto**, you must use the **uupick** command. To execute **uupick**, enter the following command:

```
uupick
```

The **uupick** program searches the public directory for any files sent to you. If it finds any, it responds with the following prompt:

```
from source_computer: file filename ?
```

The *source_computer* is the name of the sender's computer and *filename* is the name of the file transferred. In the example above, if the **uuto** transfer to *cindy* on *machine2* is successful, *cindy* sees the following **uupick** prompt:

```
from machine1: file transfile ?
```

Several options are available for responding to the **uupick** prompt. Two of the most useful are **m** [*dir*] and **d**. The **m** [*dir*] option tells **uupick** to move the file to directory *dir*. Once in *dir*, you can manipulate the file as you would any other file on your system. In the example above, *cindy* could enter the following in response to the **uupick** prompt:

```
m $HOME
```

This causes *transfile* to be moved from the public directory to *cindy*'s home directory. If no directory is specified after **m**, the file is moved to the recipient's current directory.

Entering **d** at the **uupick** prompt causes the file to be deleted from the public directory. You can quit **uupick** by entering **q**. Note other **uupick** options are available. Refer to **uupick(C)** for a complete list of these.

5

Executing Commands with **uux**

The **uux** command is used to execute commands on remote UUCP sites and on files gathered from remote UUCP sites. For security reasons, the commands available for remote execution on a computer are often very limited. A computer's */usr/lib/uucp/Permissions* file lists the commands that can be executed remotely on that computer. If you attempt to execute a command not listed in this file, you will receive mail indicating that the command cannot be executed on the computer in question.

The syntax of **uux** is:

```
uux [options] command-line
```

The *command-line* argument looks like any other UNIX command line, with the exception that commands and filenames may be prefixed with *site-name!*.

Using UUCP

The following is an example of how to execute a command on a remote system. The command causes */tmp/printfile* on *machine2* to be sent to *machine2's* default printer:

```
uux machine2!lp machine2!/tmp/printfile
```

Note that prefixing a site name to a command causes the command to be executed on that site.

The following is an example of how to execute a command on a local system on files gathered with **uux** from remote systems. Suppose that your local computer is connected to both *machine2* and *machine3*. Suppose also that you want to compare the contents of */tmp/chpt1* on *machine2* with */tmp/chpt1* on *machine3*. To do so, enter the following command:

```
uux "diff machine2!/tmp/chpt1 machine3!/tmp/chpt1 > diff.file"
```

This command will compare the contents of the files on *machine2* and *machine3* and place the output in *diff.file* in the current directory on the local computer. Since there is no site name prefixed to the **diff** command, the command is executed locally.

Note that, in the example above, the **uux** command line is placed in quotation marks. This is because it contains the redirect symbol (**>**). In general, place the **uux** command line in quotation marks whenever the command line contains special shell characters such as **<**, **>**, **|**, and so forth.

Logging in to Remote Systems

The **ct** command connects your system to a remote terminal with a modem attached. The **cu** command connects your system to a remote system. The remote system can be attached via phone lines or via a simple serial line. These commands differ from the Micnet commands and the UUCP commands discussed above in that your session with the remote system is *interactive*. The remote system “sees” you as just another user on the system. Both **ct** and **cu** are discussed below.

Using **ct**

The **ct** command connects a local computer to a remote terminal equipped with a modem and allows a user on that terminal to log in to the computer. To do this, the command dials the phone number of the remote modem. The remote modem must be able to answer the call automatically. When **ct** detects that the call has been answered, it issues a **getty** (login) process for the remote terminal and allows a user on the terminal to log in on the computer.

5

This command is especially useful when issued from the opposite end, that is, from the remote terminal itself. If you are using a remote terminal and you want to avoid long distance charges, you can use **ct** to have the computer place a call to your terminal. To do so, simply call the computer, log in, and issue the **ct** command. The computer will hang up the line and call your terminal back.

If **ct** cannot find an available dialer, it tells you that all dialers are busy and asks if it should wait until one becomes available. If you answer yes, it asks how long (in minutes) it should wait. If you answer no, **ct** quits.

The syntax of **ct** is:

```
ct [options] telno
```

The argument *telno* is the telephone number of the remote terminal.

Logging in to Remote Systems

As an example, suppose that you have a terminal with a modem attached at home and that you want to log in to the computer at work from this terminal. To avoid long distance charges, first call your work computer and log in. Then issue the `ct` command to make the computer hang up and call your terminal back. If your phone number is 932-3497, the `ct` command is:

```
ct -s1200 9323497
```

The `-s` option tells `ct` to call the modem at 1200 baud. If no device is available on the computer at work, you see the following message after executing `ct`:

```
The one 1200 baud dialer is busy
Do you want to wait for dialer? (y for yes):
```

If you type `n` (no), the `ct` command exits. If you type `y` (yes), `ct` prompts you to specify how long `ct` should wait:

```
Time, in minutes?
```

If a dialer is available when you enter the `ct` command, you see the following message:

```
Allocated dialer at 1200 baud
```

This means that a dialer has been found. You are then asked if you want the line connecting your remote terminal to the computer to be dropped:

```
Proceed to hang-up? (y to hang-up, otherwise exit):
```

Since you want to avoid long-distance charges by having the computer call you, answer `y` (yes). You are then logged off and `ct` calls your remote terminal back.

As another example, suppose that you are logged in on a computer through a local terminal and that you want to connect a remote terminal

to the computer. The phone number of the modem on the remote terminal is 932-3497. To connect the terminal, enter the following command:

```
nohup ct -h -s1200 9323497 &
```

The **-h** option tells **ct** not to disconnect the local terminal (the terminal on which the command was issued) from the computer. After the command is executed, a login prompt is displayed on the remote terminal. The user can then log in and work on the computer just as on a local terminal.

Several options are available for **ct**. Refer to **ct(C)** for a complete list of these options.

Using **cu**

The **cu** command connects your local computer to a remote computer and allows you to be logged in on both computers simultaneously. The remote computer does not have to be a UNIX system.

If the remote computer is a UNIX system, **cu** allows you to move back and forth between the two computers, transferring files and executing commands on both. Note that **cu** only allows you to transfer text files. You cannot transfer binary files with **cu**. To transfer binary files to a remote UNIX system, use either **rcp** or **uucp**.

5

The syntax of the **cu** command is:

```
cu [options] target
```

The *target* argument can take one of three forms:

phone number

This is the number of the remote computer to which you want to connect. You can embed equal signs, which represent secondary dial tones, and dashes, which represent four-second delays, in the phone number. A sample phone number might be **4084551222--341**. This number contains an area code and number, two dashes for an eight second delay and an extension.

Logging in to Remote Systems

- system-name** This is the name of a system that is listed in the */usr/lib/uucp/Systems* file. The **cu** command obtains the telephone number and the baud rate of *system-name* from this file. The **-s**, **-n**, and **-l** options should not be used with *system-name*. To see the list of computers in the *Systems* file, enter: **uname**.
- l line** This is the device name of the serial line connected to the remote computer. It has the form *ttyXX*, where *XX* is the number of a serial line.
- l line dir** Connects directly with serial line instead of making a phone connection.

Several options are available for use with the **cu** command. Refer to **cu(C)** for a complete list of these options.

Once the connection is made, if the remote computer is a UNIX system, you are presented with a login prompt. Log in as you would if you were connected locally. When you finish working on the remote computer, log off as you would if you were connected locally. Then terminate the **cu** connection by entering a tilde followed by a period (~.). You are still logged in on the local computer.

As an example, suppose that you want to log in to a remote UNIX computer via the phone lines. Suppose also that the remote computer's number is 847-7867. To connect to the remote computer, enter the following command:

```
cu -s1200 8477867
```

The **-s1200** option causes **cu** to use a 1200 baud dialer. If the **-s** option is not specified, **cu** uses the first available dialer at the speed specified in the *Devices* file.

When the remote UNIX system answers the call, **cu** notifies you that the connection has been made by displaying the following message:

```
Connected
```

Next, you are prompted for your login:

```
login:
```

Enter your login and password. Once you enter this information, you can use this computer as if you were logged in locally. When you are finished, logout and then enter:

```
~.
```

This terminates the **cu** session.

cu Command Strings

Several “Command Strings” are available with **cu** that allow your local computer to communicate with a remote UNIX system. Two of the most useful are **take** and **put**.

The **take** command allows you to copy files from the remote computer to the local computer. Suppose, for example, that you want to copy a file named *proposal* in the current directory of the remote computer to your home directory on the local computer. To do so, enter the following command:

```
~%take proposal $home/proposal
```

Note that you have to prefix a tilde and a percent sign (~%) to the **take** command, and that the tilde must be placed at the start of a line. For this reason, it is a good idea to press ⟨Return⟩ before using **take**.

The **put** command allows you to do the opposite of **take**. It copies files from the local computer to the remote computer. Suppose, for example, that you want to copy a file named *minutes* from your home directory on the local computer to the */tmp* directory of the remote computer. Suppose

Logging in to Remote Systems

also that you want the file to be called *minutes.9-18* on the remote computer. To do so, enter the following command:

```
~%put $home/minutes /tmp/minutes.9-18
```

Like the **take** command, you have to prefix a tilde and a percent sign (~%) to the **put** command, with the tilde coming at the beginning of a line. Note also that **take** and **put** copy only text files, and only to UNIX systems. They do not copy binary files.

Note

The **cu** command cannot detect or correct transmission errors. After a file transfer, you can check for loss of data by running the **sum** command on both the file that was sent and the file that was received. This command reports the total number of bytes in each file. If the totals match, your transfer was probably successful. See the **sum(C)** manual page for details.

Other command strings are available for use with **cu**. For a complete list of these, see **cu(C)**.

Chapter 6

bc: A Calculator

Introduction 6-1

Demonstration 6-2

Tasks 6-5

- Computing with Integers 6-5
- Specifying Input and Output Bases 6-6
- Scaling Quantities 6-8
- Using Functions 6-9
- Using Subscripted Variables 6-11
- Using Control Statements: if, while and for 6-11
- Using Other Language Features 6-14

Language Reference 6-16

- Tokens 6-16
- Expressions 6-17
- Function Calls 6-18
- Unary Operators 6-19
- Multiplicative Operators 6-19
- Additive Operators 6-20
- Assignment Operators 6-20
- Relational Operators 6-21
- Storage Classes 6-21
- Statements 6-22

Introduction

`bc` is a program that can be used as an arbitrary precision arithmetic calculator. `bc` 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.

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 `<CTL>d`. Once you have entered **bc**, you can use it very much like a normal calculator. As with the UNIX 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 `<BKSP>` and **INTERRUPT** (also known as ``) 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 next part of this demonstration shows you how to store values in special alphabetic registers. For example, enter:

$$a=100 ; b=5$$

Demonstration

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 UNIX 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
```

The last part of this demonstration shows you how the result of the most recent calculation is automatically stored.

The period character (.) represents the register that contains this value. It can be used in any expression, and is reset automatically whenever a new operation is performed. For example, after entering:

```
5*10
```

the `.` register contains the number 50. If you now enter:

```
+.20,
```

the result is:

```
70
```

and the `.` register now contains this new value.

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

```
quit
```

or press `<CTL>d`.

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

Tasks

This section describes how to perform common **bc** tasks. Mastery of these tasks should turn you into a competent **bc** user.

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.

Tasks

Thus the following two expressions:

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 the section on "Scaling"). For example, the lines:

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

produce the printed result:

13

Specifying Input and Output Bases

There are special internal quantities in **bc**, called **ibase** (or **base**) and **obase**. **base** and **ibase** can be used interchangeably. **ibase** is initially set to 10, and determines the base used for interpreting numbers that are read by **bc**. For example, the lines:

```
kbase = 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.

Tasks

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.

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.

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.

Tasks

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*.

Using Subscripted Variables

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

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

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

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

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

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

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

Using Control Statements: if, while and for

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

Tasks

```
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
    }
}
```

Tasks

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
$x=y=z$	$x=(y=z)$
$x=+y$	$x=x+y$
$x=-y$	$x=x-y$
$x=*y$	$x=x*y$
$x=/y$	$x=x/y$
$x=%y$	$x=x\%y$
$x=\hat{y}$	$x=x\hat{y}$
$x++$	$(x=x+1)-1$
$x--$	$(x=x-1)+1$
$++x$	$x=x+1$
$--x$	$x=x-1$

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.

Language Reference

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

Tokens

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

Comments Comments are introduced by 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 lower-case letters. Array identifiers are followed by square brackets, enclosing an optional `g`xpression 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.

Expressions

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

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

There are several types of expressions:

Named expressions

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 radices respectively. Both *ibase* and *obase* have initial values of 10.

Language Reference

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 the next section.

Function Calls

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

function-name ([*expression* [, *expression* ...]])

A whole array passed as an argument is specified by the array name followed by empty square brackets. All function arguments are passed by value. As a result, changes made to the formal parameters have no effect on the actual arguments. If the function terminates by executing a return statement, the value of the function is the value of the expression in the parentheses of the return statement, or 0 if no expression is provided or if there is no return statement. Three built-in functions are listed below:

- | | |
|--------------------------------------|---|
| <i>sqrt</i> (<i>expr</i>) | 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 <i>scale</i> , whichever is larger. |
| <i>length</i> (<i>expr</i>) | The result is the total number of significant decimal digits in the expression. The scale of the result is zero. |
| <i>scale</i> (<i>expr</i>) | The result is the scale of the expression. The scale of the result is zero. |

Unary Operators

The unary operators bind right to left.

- | | |
|----------------------------|--|
| <code>- expr</code> | The result is the negative of the expression. |
| <code>++ named_expr</code> | The named expression is incremented by one. The result is the value of the named expression after incrementing. |
| <code>-- named_expr</code> | The named expression is decremented by one. The result is the value of the named expression after decrementing. |
| <code>named_expr ++</code> | The named expression is incremented by one. The result is the value of the named expression before incrementing. |
| <code>named_expr --</code> | The named expression is decremented by one. The result is the value of the named expression before decrementing. |

Multiplicative Operators

The multiplicative operators (`*`, `/`, and `%`) bind from left to right.

- | | |
|------------------------|--|
| <code>expr*expr</code> | 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:

$\min(a+b, \max(\text{scale}, a, b))$ |
| <code>expr/expr</code> | The result is the quotient of the two expressions. The scale of the result is the value of <i>scale</i> . |
| <code>expr%expr</code> | The modulo operator (<code>%</code>) 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 <i>scale</i> . |
| <code>expr^expr</code> | 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:

$$\min(a*b, \max(\text{scale}, a))$$

Additive Operators

The additive operators bind left to right.

expr+expr The result is the sum of the two expressions. The scale of the result is the maximum of the scales of the expressions.

expr-expr The result is the difference of the two expressions. The scale of the result is the maximum of the scales of the expressions.

Assignment Operators

The assignment operators listed below assign values to the named expression on the left side.

named_expr=expr
This expression results in assigning the value of the expression on the right to the named expression on the left.

named_expr=+expr
The result of this expression is equivalent to *named_expr=named_expr+expr*.

named_expr=-expr
The result of this expression is equivalent to *named_expr=named_expr-expr*.

*named_expr=*expr*
The result of this expression is equivalent to *named_expr=named_expr*expr*.

named_expr=/expr
The result of this expression is equivalent to *named_expr=named_expr/expr*.

named_expr=%expr

The result of this expression is equivalent to *named_expr=named_expr%expr*.

named_expr=^expr

The result of this expression is equivalent to *named_expr=named_expr^expr*.

Relational Operators

Unlike all other operators, the relational operators are only valid as the object of an **if** or **while** statement, or inside a **for** statement.

These operators are listed below:

expr < expr

expr > expr

expr <= expr

expr >= expr

expr == expr

expr != expr

Storage Classes

6

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 name, followed by empty square brackets.

Automatic variables in **bc** do not work the same way as in C. On entry to a function, the old values of the names that appear as parameters and as automatic variables are pushed onto a stack. Until return is made from the function, reference to these names refers only to the new values.

Statements

Statements must be separated by a semicolon or a newline. Except where altered by control statements, execution is sequential. There are four types of statements: expression statements, compound statements, quoted string statements, and built-in statements. Each kind of statement is discussed below:

Expression statements

When a statement is an expression, unless the main operator is an assignment, the value of the expression is printed, followed by a newline character.

Compound statements

Statements can be grouped together and used when one statement is expected by surrounding them with curly braces ({ 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 syntax is as follows:

```
if (relation) statement
```

Quit statement

The **quit** statement stops execution of a **bc** program and returns control to the Operating System 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 $\langle\text{CTL}\rangle$ 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
```

Chapter 7

The Shell

Introduction 7-1

Basic Concepts 7-2

- How Shells Are Created 7-2
- Commands 7-2
- How the Shell Finds Commands 7-3
- Generation of Argument Lists 7-3
- Quoting Mechanisms 7-4
- Standard Input and Output 7-6
- Diagnostic and Other Outputs 7-7
- Command Lines and Pipelines 7-7
- Command Substitution 7-9

Shell Variables 7-11

- Positional Parameters 7-11
- User-Defined Variables 7-12
- Predefined Special Variables 7-16

The Shell State 7-18

- Changing Directories 7-18
- The .profile File 7-19
- Execution Flags 7-19

A Command's Environment 7-20

Invoking the Shell 7-22

Passing Arguments to Shell Procedures 7-23

Controlling the Flow of Control 7-26

- Using the if Statement 7-28
- Using the case Statement 7-29
- Conditional Looping: while and until 7-30
- Looping Over a List: for 7-31
- Loop Control: break and continue 7-32
- End-of-File and exit 7-33
- Command Grouping: Parentheses and Braces 7-33

- Defining Functions 7-35
- Input/Output Redirection and Control Commands 7-36
- Transfer Between Files: The Dot (.) Command 7-36
- Interrupt Handling: trap 7-36

Special Shell Commands 7-40

Creation and Organization of Shell Procedures 7-44

More About Execution Flags 7-46

Supporting Commands and Features 7-47

- Conditional Evaluation: test 7-47

- Echoing Arguments 7-49

- Expression Evaluation: expr 7-49

- True and False 7-50

- In-Line Input Documents 7-50

- Input / Output Redirection Using File Descriptors 7-51

- Conditional Substitution 7-52

- Invocation Flags 7-54

Effective and Efficient Shell Programming 7-55

- Number of Processes Generated 7-55

- Number of Data Bytes Accessed 7-57

- Shortening Directory Searches 7-58

- Directory-Search Order and the PATH Variable 7-58

- Good Ways to Set Up Directories 7-59

Shell Procedure Examples 7-60

Shell Grammar 7-68

Introduction

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

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

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

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

Basic Concepts

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

How Shells Are Created

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

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

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

Commands

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

```
lpr allan barry calvin
```

If the first argument of a command names a file that is *executable* (as indicated by an appropriate set of permission bits associated with that file) and is actually a compiled program, the shell, as parent, creates a child process that immediately executes that program. If the file is marked as being executable, but is not a compiled program, it is assumed

to be a shell procedure, that is, a file of ordinary text containing shell command lines. In this case, the shell spawns another instance of itself (a *subshell*) to read the file and execute the commands inside it.

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

How the Shell Finds Commands

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

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

7

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.

Basic Concepts

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

Examples of metacharacter usage:

Metacharacter	Meaning
*	Matches all names in the current directory
temp	Matches all names containing "temp"
[a-f]*	
*.c	
/usr/bin/?	Matches all single-character names in /usr/bin

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

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

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

* ? []

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

Quoting Mechanisms

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

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

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

results in the string:

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

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

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

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

Some examples of quoting are displayed below:

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

Standard Input and Output

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

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

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

```
> output
```

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

```
who >> log
```

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

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

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

```
cat < ?
```

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

Diagnostic and Other Outputs

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

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

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

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

```
cmd 9> savedata
```

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

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

Command Lines and Pipelines

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

Basic Concepts

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

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

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

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

- **who**
Prints the list of logged-in users on the terminal screen.
- **who >>log**
Appends the list of logged-in users to the end of file *log*.
- **who | wc -l**
Prints the number of logged-in users. (The argument to **wc** is pronounced “minus ell”.)
- **who | pr**
Prints a paginated list of logged-in users.
- **who | sort**
Prints an alphabetized list of logged-in users.
- **who | grep bob**
Prints the list of logged-in users whose login names contain the string *bob*.
- **who | grep bob | sort | pr**
Prints an alphabetized, paginated list of logged-in users whose login names contain the string *bob*.

- `{ date; who | wc -l; } >> log`
Appends (to file *log*) the current date followed by the count of logged-in users. Be sure to place a space after the left brace and a semicolon before the right brace.
- `who | sed -e 's/ .*//' | sort | uniq -d`
Prints only the login names of all users who are logged in more than once. Note the use of `sed` as a filter to remove characters trailing the login name from each line. (The “`.*`” in the `sed` command is preceded by a space.)

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

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

is the same as:

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

Command Substitution

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

For example:

```
today=`date`
```

Basic Concepts

assigns the string representing the current date to the variable “today”; for example “Tue Nov 26 16:01:09 EST 1985”. The following command saves the number of logged-in users in the shell variable *users* :

```
users=`who | wc -l`
```

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

```
logmsg=`echo Your login directory is `pwd``
```

will display the line “your login directory is *name of login directory*”. Shell variables can also be given values indirectly by using the **read** and **line** commands. The **read** command takes a line from the standard input (usually your terminal) and assigns consecutive words on that line to any variables named.

For example:

```
read first init last
```

takes an input line of the form:

```
G. A. Snyder
```

and has the same effect as entering:

```
first=G. init=A. last=Snyder
```

The **read** command assigns any excess “words” to the last variable.

The **line** command reads a line of input from the standard input and then echoes it to the standard output.

Shell Variables

The shell has several mechanisms for creating variables. A variable is a name representing a string value. Certain variables are referred to as *positional parameters*; these are the variables that are normally set only on the command line. Other shell variables are simply names to which the user or the shell itself may assign string values.

Positional Parameters

When a shell procedure is invoked, the shell implicitly creates *positional parameters*. The name of the shell procedure itself in position zero on the command line is assigned to the positional parameter \$0. The first command argument is called \$1, and so on. The `shift` command may be used to access arguments in positions numbered higher than nine. For example, the following shell script might be used to cycle through command line switches and then process all succeeding files:

```
while test -n "$1"
do case $1 in
    -a) A=aooption ; shift ;;
    -b) B=booption ; shift ;;
    -c) C=cooption ; 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.

Shell Variables

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 <code>login</code> program to the name of the user's <i>login directory</i> , that is, the directory that becomes the current directory upon completion of a login; <code>cd</code> without arguments switches to the <code>\$HOME</code> 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 <i>internal field separators</i> . These are the characters the shell uses during blank interpretation. (If you want to parse some delimiter-separated data easily, you can set <code>IFS</code> to include that delimiter.)

Shell Variables

The shell initially sets IFS to include the blank, tab, and newline characters.

- MAIL** The pathname of a file where your mail is deposited. If MAIL is set, then the shell checks to see if anything has been added to the file it names and announces the arrival of new mail each time you return to command level (e.g., by leaving the editor). MAIL is not set automatically; if desired, it should be set (and optionally “exported”) in the user’s *.profile*. (The **export** command and *.profile* file are discussed later in this chapter.) (The presence of mail in the standard mail file is also announced at login, regardless of whether MAIL is set.)
- MAILCHECK** This parameter specifies how often (in seconds) the shell will check for the arrival of mail in the files specified by the MAILPATH or MAIL parameters. The default value is 600 seconds (10 minutes). If set to 0, the shell will check before each prompt.
- MAILPATH** A colon (:) separated list of file names. If this parameter is set, the shell informs the user of the arrival of mail in any of the specified files. Each file name can be followed by % and a message that will be printed when the modification time changes. The default message is *you have mail*.
- SHACCT** If this parameter is set to the name of a file writable by the user, the shell will write an accounting record in the file for each shell procedure executed. Accounting routines such as **acctcom**(ADM) and **accton**(ADM) can be used to analyze the data collected.
- SHELL** When the shell is invoked, it scans the environment for this name. If it is found and there is an ‘r’ in the file name part of its value, the shell becomes a restricted shell.
- PATH** The variable that specifies the search path used by the shell in finding commands. Its value is an ordered list of directory pathnames separated by colons. The shell initializes PATH to the list *:/bin:/usr/bin* where a null argument appears in front of the first colon. A null anywhere in the

path list represents the current directory. On some systems, a search of the current directory is *not* the default and the PATH variable is initialized instead to `/bin:/usr/bin`. If you wish to search your current directory last, rather than first, use:

```
PATH=/bin:/usr/bin:
```

Below, the two colons together represent a colon followed by a null, followed by a colon, thus naming the current directory. You could possess a personal directory of commands (say, `$HOME/bin`) and cause it to be searched *before* the other three directories by using:

```
PATH=$HOME/bin:./bin:/usr/bin
```

PATH is normally set in your *.profile* file.

CDPATH

This variable defines the search path for the directory containing `arg`. Alternative directory names are separated by a colon (:). The default path is `<null>` (specifying the current directory). The current directory is specified by a null path name, which can appear immediately after the equal sign or between the colon delimiters anywhere else in the path list. If `arg` begins with a / then the search path is not used. Otherwise, each directory in the path is searched for `arg`.

PS1

The variable that specifies what string is to be used as the primary *prompt* string. If the shell is interactive, it prompts with the value of PS1 when it expects input. The default value of PS1 is `"$ "` (a dollar sign (\$) followed by a blank).

PS2

The variable that specifies the secondary prompt string. If the shell expects more input when it encounters a newline in its input, it prompts with the value of PS2. The default value for this variable is `"> "` (a greater-than symbol followed by a space).

Shell Variables

In general, you should be sure to **export** all of the above variables so that their values are passed to all shells created from your login. Use **export** at the end of your *.profile* file. An example of an **export** statement follows:

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

Predefined Special Variables

Several variables have special meanings; the following are set *only* by the shell:

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

```
sh cmd a b c
```

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

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

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

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

The following example illustrates the recommended practice of creating temporary files; note that the directories */usr* and */usr/tmp* are cleared out if the system is rebooted.

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

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

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

The Shell State

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

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

Changing Directories

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

For example, the first sequence below copies the file */etc/passwd* to */usr/you/passwd*; the second example first changes directory to */etc* and then copies the file:

```
cp /etc/passwd /usr/you/passwd
(cd /etc; cp passwd /usr/you/passwd)
```

Note the use of parentheses. Both command lines have the same effect.

If the shell is reading its commands from a terminal, and the specified directory does not exist (or some component cannot be searched), spelling correction is applied to each component of *directory*, in a search for the “correct” name. The shell then asks whether or not to try and change directory to the corrected directory name; an answer of *n* means “no”, and anything else is taken as “yes.”

The .profile File

The file named *.profile* is read each time you log in. It is normally used to execute special one-time-only commands and to set and export variables to all later shells. Only after commands are read and executed from *.profile*, does the shell read commands from the standard input—usually the terminal.

If you wish to reset the environment after making a change to the *.profile* file, enter

```
.profile
```

This command eliminates the need to log out and then log in again to execute *.profile*.

Execution Flags

The `set` command lets you alter the behavior of the shell by setting certain shell flags. In particular, the `-x` and `-v` flags may be useful when invoking the shell as a command from the terminal. The flags `-x` and `-v` may be set by entering:

```
set -xv
```

The same flags may be turned *off* by entering:

```
set +xv
```

These two flags have the following meaning:

- v Input lines are printed as they are read by the shell. This flag is particularly useful for isolating syntax errors. The commands on each input line are executed after that input line is printed.
- x Commands and their arguments are printed as they are executed. (Shell control commands, such as `for`, `while`, etc., are not printed, however.) Note that `-x` causes a trace of only those commands that are actually executed, whereas `-v` prints each line of input until a syntax error is detected.

The `set` command is also used to set these and other flags within shell procedures.

A Command's Environment

All variables and their associated values that are known to a command at the beginning of its execution make up its *environment*. This environment includes variables that the command inherits from its parent process and variables specified as *keyword parameters* on the command line that invokes the command.

The variables that a shell passes to its child processes are those that have been named as arguments to the **export** command. The **export** command places the named variables in the environments of both the shell *and* all its future child processes.

Keyword parameters are variable-value pairs that appear in the form of assignments, normally *before* the procedure name on a command line. Such variables are placed in the environment of the procedure being invoked. For example:

```
# keycommand
echo $a $b
```

This is a simple procedure that echoes the values of two variables. If it is invoked as:

```
a=key1 b=key2 keycommand
```

then the resulting output is:

```
key1 key2
```

Keyword parameters are *not* counted as arguments to the procedure and do not affect \$#.

A procedure may access the value of any variable in its environment. However, if changes are made to the value of a variable, these changes are not reflected in the environment; they are local to the procedure in question. In order for these changes to be placed in the environment that the procedure passes to *its* child processes, the variable must be named as an argument to the **export** command within that procedure. To obtain a list of variables that have been made exportable from the current shell, enter:

```
export
```

You will also get a list of variables that have been made **readonly**. To get a list of name-value pairs in the current environment, enter either:

`printenv`

or

`env`

Invoking the Shell

The shell is a command and may be invoked in the same way as any other command:

`sh proc [arg ...]`

A new instance of the shell is explicitly invoked to read *proc*. Arguments, if any, can be manipulated.

`sh -v proc [arg ...]`

This is equivalent to putting “set -v” at the beginning of *proc*. It can be used in the same way for the -x, -e, -u, and -n flags.

`proc [arg ...]`

If *proc* is an executable file, and is not a compiled executable program, the effect is similar to that of:

`sh proc args`

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

Passing Arguments to Shell Procedures

When a command line is scanned, any character sequence of the form `$n` is replaced by the *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
```

Passing Arguments to Shell Procedures

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
```

Passing Arguments to Shell Procedures

This segment of code results in the execution of the command line:

```
who | wc -l
```

Uses of `eval` can be nested so that a command line can be evaluated several times.

Controlling the Flow of Control

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

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

A *command* is a simple command or any of the shell control commands described below. A *pipeline* is a sequence of one or more commands separated by vertical bars (`|`). In a pipeline, the standard output of each command but the last is connected (by a *pipe*) to the standard input of the next command. Each command in a pipeline is run separately; the shell waits for the last command to finish. The exit status of a pipeline is the exit status of last process in the pipeline.

A *command list* is a sequence of one or more pipelines separated by a semicolon (`;`), an ampersand (`&`), an “and-if” symbol (`&&`), or an “or-if” (`||`) symbol, and optionally terminated by a semicolon or an ampersand. A semicolon causes sequential execution of the previous pipeline. This means that the shell waits for the pipeline to finish before reading the next pipeline. On the other hand, the ampersand (`&`) causes asynchronous background execution of the preceding pipeline. Thus, both sequential and background execution are allowed. A background pipeline continues execution until it terminates voluntarily, or until its processes are killed.

Other uses of the ampersand include off-line printing, background compilation, and generation of jobs to be sent to other computers. For example, if you enter:

```
nohup cc prog.c&
```

You may continue working while the C compiler runs in the background. A command line ending with an ampersand is immune to interrupts or quits that you might generate by typing `INTERRUPT` or `QUIT`. However, `<CTL>d` will abort the command if you are operating over a dial-up line or have `stty hupcl`. In this case, it is wise to make the command immune to hang-ups (i.e., logouts) as well. The `nohup` command is used for this purpose. In the above example without `nohup`, if you log out from a dial-up line while `cc` is still executing, `cc` will be killed and your output will disappear.

The ampersand operator should be used with restraint, especially on heavily-loaded systems. Other users will not consider you a good citizen if you start up a large number of background processes without a compelling reason for doing so.

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

```
cmd1 || cmd2
```

the first command, *cmd1*, is executed and its exit status examined. Only if *cmd1* fails (i.e., has a nonzero exit status) is *cmd2* executed. Thus, this is a more terse notation for:

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

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

```
cmd1 && cmd2
```

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

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

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

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

Note that a space is needed after the left brace and that a semicolon should appear before the right brace.

Using the `if` Statement

The shell provides structured conditional capability with the `if` command. The simplest `if` command has the following form:

```
if command-list
then command-list
fi
```

The command list following the `if` is executed and if the last command in the list has a zero exit status, then the command list that follows `then` is executed. The word `fi` indicates the end of the `if` command.

To cause an alternative set of commands to be executed when there is a nonzero exit status, an `else` clause can be given with the following structure:

```
if command-list
then command-list
else command-list
fi
```

Multiple tests can be achieved in an `if` command by using the `elif` clause, although the `case` statement may be better for large numbers of tests. For example:

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

The above example is executed as follows: if the value of the first positional parameter is a filename (`-f`), then print that file; if not, then check to see if it is the name of a directory (`-d`). If so, change to that directory (`cd`) and print all the files there (`pr *`). Otherwise, `echo` the error message.

The `if` command may be nested (but be sure to end each one with a `fi`). The newlines in the above examples of `if` may be replaced by semicolons.

The exit status of the `if` command is the exit status of the last command executed in any `then` clause or `else` clause. If no such command was executed, `if` returns a zero exit status.

Note that an alternate notation for the `test` command uses brackets to enclose the expression being tested. For example, the previous example might have been written as follows:

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

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

Using the case Statement

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

```
case string in
    pattern ) command-list ;;
    ...
    pattern ) command-list ;;
esac
```

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

Controlling the Flow of Control

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

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

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

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

Conditional Looping: while and until

A **while** command has the general form:

```
while command-list
do
    command-list
done
```

The commands in the first *command-list* are executed, and if the exit status of the last command in that list is zero, then the commands in the second *command-list* are executed. This sequence is repeated as long as the exit status of the first *command-list* is zero. A loop can be executed as long as the first command-list returns a nonzero exit status by replacing **while** with **until**.

Any newline in the above example may be replaced by a semicolon. The exit status of a **while** (or **until**) command is the exit status of the last command executed in the second *command-list*. If no such command is executed, **while** (or **until**) has a zero exit status.

Looping Over a List: for

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

```
for variable in word-list
do
    command-list
done
```

Here *word-list* is a list of strings separated by blanks. The commands in the *command-list* are executed once for each word in the *word-list*. *Variable* takes on as its value each word from the word list, in turn. The word list is fixed after it is evaluated the first time. For example, the following **for** loop causes each of the C source files *xec.c*, *cmd.c*, and *word.c* in the current directory to be compared with a file of the same name in the directory */usr/src/cmd/sh*:

```
for CFILE in xec cmd word
do diff $CFILE.c /usr/src/cmd/sh/$CFILE.c
done
```

Note that the first occurrence of **CFILE** immediately after the word **for** has no preceding dollar sign, since the name of the variable is wanted and not its value.

You can omit the “**in word-list**” part of a **for** command; this causes the current set of positional parameters to be used in place of *word-list*. This is useful when writing a command that performs the same set of commands for each of an unknown number of arguments.

As an example, create a file named *echo2* that contains the following shell script:

```
for word
do echo $word$word
done
```

Give *echo2* execute status:

```
chmod +x echo2
```

Controlling the Flow of Control

Now type the following command:

```
echo2 ma pa bo fi yo no so ta
```

The output from this command is:

```
mama  
papa  
bobo  
fifi  
yoyo  
nono  
soso  
tata
```

Loop Control: break and continue

The **break** command can be used to terminate execution of a **while** or a **for** loop. The **continue** command immediately starts the execution of the next iteration of the loop. These commands are effective only when they appear between **do** and **done**.

The **break** command terminates execution of the smallest (i.e., innermost) enclosing loop, causing execution to resume after the nearest following unmatched **done**. Exit from n levels is obtained by **break** n .

The **continue** command causes execution to resume at the nearest enclosing **for**, **while**, or **until** statement, i.e., the one that begins the innermost loop containing the **continue**. You can also specify an argument *n* to **continue** and execution will resume at the *n*th enclosing loop:

```
# This procedure is interactive.
# "Break" and "continue" commands are used
# to allow the user to control data entry.
while true #loop forever
do   echo "Please enter data"
     read response
     case "$response" in
       "done")    break
                 # no more data
                 ;;
       "")        # just a carriage return,
                 # keep on going
                 continue
                 ;;
       *)         # process the data here
                 ;;
     esac
done
```

End-of-File and exit

When the shell reaches the end-of-file in a shell procedure, it terminates execution, returning to its parent the exit status of the last command executed prior to the end-of-file. The top level shell is terminated by typing a `<CTL>d` (which logs the user out of the system).

The **exit** command simulates an end-of-file, setting the exit status to the value of its argument, if any. Thus, a procedure can be terminated normally by placing “`exit 0`” at the end of the file.

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

Controlling the Flow of Control

wherever they appear in a command line—they can appear as literal parentheses *only* when enclosed in quotation marks. For example, if you enter:

```
garble(stuff)
```

the shell prints an error message. Quoted lines, such as:

```
garble("stuff")  
"garble(stuff)"
```

are interpreted correctly. Other quoting mechanisms are discussed in “Quoting Mechanisms” in this chapter.

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

The current environment is passed to the subshell and variables that are exported in the current shell are also exported in the subshell. Thus:

```
CURRENTDIR=`pwd`; cd /usr/docs/otherdir;  
nohup nroff doc.n > doc.out&; cd $CURRENTDIR
```

and

```
(cd /usr/docs/otherdir; nohup nroff doc.n > doc.out&)
```

accomplish the same result: */usr/docs/otherdir/doc.n* is processed by *nroff* and the output is saved in */usr/docs/otherdir/doc.out*. (Note that **nroff** is a text processing command.) However, the second example automatically puts you back in your original working directory. In the second example above, blanks or newlines surrounding the parentheses are allowed but not necessary. When entering a command line at your terminal, the shell will prompt with the value of the shell variable PS2 if an end parenthesis is expected.

Braces (**{** and **}**) may also be used to group commands together. Both the left and the right brace are recognized *only* if they appear as the first (unquoted) word of a command. The opening brace may be followed by a newline (in which case the shell prompts for more input). Unlike parentheses, no subshell is created for braces: the enclosed commands are simply read by the shell. The braces are convenient when you wish to use the (sequential) output of several commands as input to one command.

The exit status of a set of commands grouped by either parentheses or braces is the exit status of the last enclosed executed command.

Defining Functions

The shell includes a function definition capability. Functions are like shell scripts or procedures except that they reside in memory and so are executed by the shell process, not by a separate process. The basic form is:

```
name ( ) {list;
```

list can include any of the commands previously discussed. Functions can be defined in one section of a shell script to be called as many times as needed, making them easier to write and maintain. Here is an example of a function called “getyn”:

```
# Prompt for yes or no answer - returns non-zero for no
getyn( ) {
    while echo "$* (y/n)? \C" >& 2
    do read yn rest
      case $yn in
        [yY]) return 0 ;;
        [nN]) return 1 ;;
        *) echo "Please answer y or n" >&2 ;;
      esac
    done
}
```

In this example, the function appends a “(y/n)?” to the output and accepts “Y”, “y”, “n” or “N” as input, returning a 0 or 1. If the input is anything else, the function prompts the user for the correct input. (Echo should never fail, so the while-loop is effectively infinite.)

Functions are used just like other commands; an invocation of *getyn* might be:

```
getyn "Do you wish to continue" || exit
```

However, unlike other commands, the shell positional parameters \$1, \$2, ..., are set to the arguments of the function. Since an exit in a function will terminate the shell procedure, the return command should be used to return a value back to the procedure.

Input/Output Redirection and Control Commands

The shell normally does *not* fork and create a new shell when it recognizes the control commands (other than parentheses) described above. However, each command in a pipeline is run as a separate process in order to direct input to or output from each command. Also, when redirection of input or output is specified explicitly to a control command, a separate process is spawned to execute that command. Thus, when **if**, **while**, **until**, **case**, and **for** are used in a pipeline consisting of more than one command, the shell forks and a subshell runs the control command. This has two implications:

1. Any changes made to variables within the control command are not effective once that control command finishes (this is similar to the effect of using parentheses to group commands).
2. Control commands run slightly slower when redirected, because of the additional overhead of creating a shell for the control command.

Transfer Between Files: The Dot (.) Command

A command line of the form:

```
. proc
```

causes the shell to read commands from *proc* without spawning a new process. Changes made to variables in *proc* are in effect after the dot command finishes. This is a good way to gather a number of shell variable initializations into one file. A common use of this command is to reinitialize the top level shell by reading the *.profile* file with:

```
. .profile
```

Interrupt Handling: trap

Shell procedures can use the **trap** command to disable a signal (cause it to be ignored), or redefine its action. The form of the **trap** command is:

```
trap arg signal-list
```

Controlling the Flow of Control

Here *arg* is a string to be interpreted as a command list and *signal-list* consists of one or more signal numbers as described in **signal (S)** in the *Programmer's Reference*. The most important of these signals follow:

Number	Signal
0	Exit from the shell
1	HANGUP
2	INTERRUPT character (DELETE or RUB OUT)
3	QUIT (<CTL>)
9	KILL (cannot be caught or ignored)
11	Segmentation violation (cannot be caught or ignored)
15	Software termination signal

The commands in *arg* are scanned at least once, when the shell first encounters the **trap** command. Because of this, it is usually wise to use single rather than double quotation marks to surround these commands. The former inhibit immediate command and variable substitution. This becomes important, for instance, when one wishes to remove temporary files and the names of those files have not yet been determined when the trap command is first read by the shell. The following procedure will print the name of the current directory in the user information as to how much of the job was done:

```
trap 'echo Directory was `pwd` when interrupted' 2 3 15
for i in /bin /usr/bin /usr/gas/bin
do
    cd $i
    # commands to be executed in directory $i here
done
```

Beware that the same procedure with double rather than single quotation marks does something different. The following prints the name of the directory from which the procedure was first executed:

```
trap "echo Directory was `pwd` when interrupted" 2 3 15
```

A signal 11 can never be trapped, because the shell itself needs to catch it to deal with memory allocation. Zero is interpreted by the **trap** command as a signal generated by exiting from a shell. This occurs either with an **exit** command, or by “falling through” to the end of a procedure. If *arg* is not specified, then the action taken upon receipt of any of the signals in the signal list is reset to the default system action. If *arg* is an explicit null string (“” or “”), then the signals in the signal list are ignored by the shell.

Controlling the Flow of Control

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 ((CTL)d) or an interrupt is received. An end-of-file causes the **read** command to return a nonzero exit status, and thus the **while** loop terminates and the next directory cycle is initiated. An interrupt is ignored while executing the requested commands, but causes termination of the procedure when it is waiting for input:

```
d=`pwd`
for i in *
do   if test -d $d/$i
      then cd $d/$i
           while echo "$i:"
                 trap exit 2
                 read x
           do   trap : 2
                 # ignore interrupts
                 eval $x
           done
      fi
done
```

Several **traps** may be in effect at the same time: if multiple signals are received simultaneously, they are serviced in numerically ascending order. To determine which traps are currently set, enter:

```
trap
```

It is important to understand some things about the way in which the shell implements the **trap** command. When a signal (other than 11) is received by the shell, it is passed on to whatever child processes are currently executing.

When these (synchronous) processes terminate, normally or abnormally, the shell polls any traps that happen to be set and executes the appropriate **trap** commands. This process is straightforward, except in the case of traps set at the command (outermost, or login) level. In this case, it is possible that no child process is running, so before the shell polls the traps, it waits for the termination of the first process spawned *after* the signal was received.

When a signal is redefined in a shell script, this does not redefine the signal for programs invoked by that script; the signal is merely passed along. A disabled signal is not passed.

For internal commands, the shell normally polls traps on completion of the command. An exception to this rule is made for the **read** command, for which traps are serviced immediately, so that **read** can be interrupted while waiting for input.

Shell Script Example

The following is a good shell script for handling signals.

```
:
#
#           Set signal handlers for shell script
#
trap "echo
trap "echo
trap "echo
#
#   Note:  If you cd to a different directory you may want to
#   reset the trap for SIGQUIT so it will find the "core" file.
#   To do this you would put the same line below the cd command
#   in the shell script.
#
trap "echo

echo " Going into loop0
while true
do
    cd /tmp
    trap "echo
    lf
    cd /usr
    trap "echo
    lf
    sleep 1
done
echo " Leaving the loop 0
exit 0
```

Special Shell Commands

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

Special Shell Commands

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 *User's Reference* for information on the value of *nnn*.

unset *name* For each *name*, remove the corresponding variable or function. The variables **PATH**, **PS1**, **PS2**, **MAILCHECK** and **IFS** cannot be unset.

`wait n`

The shell waits for all currently active child processes to terminate. If *n* is specified, the shell waits for the specified process to terminate. The exit status of *wait* is always zero if *n* is not given; otherwise it is the exit status of child *n*.

Creation and Organization of Shell Procedures

A shell procedure can be created in two simple steps. The first is building an ordinary text file. The second is changing the *mode* of the file to make it *executable*, thus permitting it to be invoked by:

```
proc args
```

rather than

```
sh proc args
```

The second step may be omitted for a procedure to be used once or twice and then discarded, but is recommended for frequently-used ones. For example, create a file named *mailall* with the following contents:

```
LETTER=$1
shift
for i in $*
do mail $i < $LETTER
done
```

Next enter:

```
chmod +x mailall
```

The new command might then be invoked from within the current directory by entering:

```
mailall letter joe bob
```

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

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

Shell procedures are often used by users running the `csh`. However, if the first character of the procedure is a `#` (comment character), the `sh` assumes the procedure is a `csh` script, and invokes `/bin/csh` to execute it. Always start `sh` procedures with some other character if `csh` users are to run the procedure at any time. This invokes the standard shell `/bin/sh`.

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

Many users prefer writing shell procedures to writing programs in C or other traditional languages. This is true for several reasons:

1. A shell procedure is easy to create and maintain because it is only a file of ordinary text.
2. A shell procedure has no corresponding object program that must be generated and maintained.
3. A shell procedure is easy to create quickly, use a few times, and then remove.
4. Because shell procedures are usually short in length, written in a high-level programming language, and kept only in their source-language form, they are generally easy to find, understand, and modify.

By convention, directories that contain only commands and shell procedures are named `bin`. This name is derived from the word “binary”, and is used because compiled and executable programs are often called “binaries” to distinguish them from program source files. Most groups of users sharing common interests have one or more `bin` directories set up to hold common procedures. Some users have their `PATH` variable list several such directories. Although you can have a number of such directories, it is unwise to go overboard: it may become difficult to keep track of your environment and efficiency may suffer.

More About Execution Flags

There are several execution flags available in the shell that can be useful in shell procedures:

- e This flag causes the shell to exit immediately if any command that it executes exits with a nonzero exit status. This flag is useful for shell procedures composed of simple command lines; it is not intended for use in conjunction with other conditional constructs.

- u This flag causes unset variables to be considered errors when substituting variable values. This flag can be used to effect a global check on variables, rather than using conditional substitution to check each variable.

- t This flag causes the shell to exit after reading and executing the commands on the remainder of the current input line. This flag is typically used by C programs which call the shell to execute a single command.

- n This is a “don’t execute” flag. On occasion, one may want to check a procedure for syntax errors, but not execute the commands in the procedure. Using “set -nv” at the beginning of a file will accomplish this.

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

Supporting Commands and Features

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

Conditional Evaluation: `test`

The `test` command evaluates the expression specified by its arguments and, if the expression is true, `test` returns a zero exit status. Otherwise, a nonzero (false) exit status is returned. `test` also returns a nonzero exit status if it has no arguments. Often it is convenient to use the `test` command as the first command in the command list following an `if` or a `while`. Shell variables used in `test` expressions should be enclosed in double quotation marks if there is any chance of their being null or not set.

The square brackets may be used as an alias to `test`, so that:

[*expression*]

has the same effect as:

`test expression`

Note that the spaces before and after the *expression* in brackets are essential.

The following is a partial list of the options that can be used to construct a conditional expression:

<code>-r <i>file</i></code>	True if the named file exists and is readable by the user.
<code>-w <i>file</i></code>	True if the named file exists and is writable by the user.
<code>-x <i>file</i></code>	True if the named file exists and is executable by the user.
<code>-s <i>file</i></code>	True if the named file exists and has a size greater than zero.

Supporting Commands and Features

<code>-d file</code>	True if the named file is a directory.
<code>-f file</code>	True if the named file is an ordinary file.
<code>-z s1</code>	True if the length of string <i>s1</i> is zero.
<code>-n s1</code>	True if the length of the string <i>s1</i> is nonzero.
<code>-t fildes</code>	True if the open file whose file descriptor number is <i>fildes</i> is associated with a terminal device. If <i>fildes</i> is not specified, file descriptor 1 is used by default.
<code>s1 = s2</code>	True if strings <i>s1</i> and <i>s2</i> are identical.
<code>s1 != s2</code>	True if strings <i>s1</i> and <i>s2</i> are <i>not</i> identical.
<code>s1</code>	True if <i>s1</i> is <i>not</i> the null string.
<code>n1 -eq n2</code>	True if the integers <i>n1</i> and <i>n2</i> are algebraically equal; other algebraic comparisons are indicated by <code>-ne</code> (not equal), <code>-gt</code> (greater than), <code>-ge</code> (greater than or equal to), <code>-lt</code> (less than), and <code>-le</code> (less than or equal to).

These may be combined with the following operators:

<code>!</code>	Unary negation operator.
<code>-a</code>	Binary logical AND operator.
<code>-o</code>	Binary logical OR operator; it has lower precedence than the logical AND operator (<code>-a</code>).
<code>(expr)</code>	Parentheses for grouping; they must be escaped to remove their significance to the shell. In the absence of parentheses, evaluation proceeds from left to right.

Note that all options, operators, filenames, etc. are separate arguments to `test`.

Echoing Arguments

The `echo` command has the following syntax:

```
echo [ options ] [ args ]
```

`echo` copies its arguments to the standard output, each followed by a single space, except for the last argument, which is normally followed by a newline. You can use it to prompt the user for input, to issue diagnostics in shell procedures, or to add a few lines to an output stream in the middle of a pipeline. Another use is to verify the argument list generation process before issuing a command that does something drastic.

You can replace the `ls` command with

```
echo *
```

because the latter is faster and prints fewer lines of output.

The `-n` option to `echo` removes the newline from the end of the echoed line. Thus, the following two commands prompt for input and then allow entering on the same line as the prompt:

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

The `echo` command also recognizes several escape sequences described in `echo (C)` in the *User's Reference*.

Expression Evaluation: `expr`

The `expr` command provides arithmetic and logical operations on integers and some pattern-matching facilities on its arguments. It evaluates a single expression and writes the result on the standard output; `expr` can be used inside grave accents to set a variable. Some typical examples follow:

```
#      increment $A
A=`expr $a + 1`
#      put third through last characters of
#      $1 into substring
substring=`expr "$1" : '..\(.*)'`
#      obtain length of $1
c=`expr "$1" : '.*'`
```

Supporting Commands and Features

The most common uses of **expr** are in counting iterations of a loop and in using its pattern-matching capability to pick apart strings.

True and False

The **true** and **false** commands perform the functions of exiting with zero and nonzero exit status, respectively. The **true** and **false** commands are often used to implement unconditional loops. For example, you might enter:

```
while true
do echo forever
done
```

This will echo “forever” on the screen until an INTERRUPT is entered.

In-Line Input Documents

Upon seeing a command line of the form:

```
command << eofstring
```

where *eofstring* is any arbitrary string, the shell will take the subsequent lines as the standard input of *command* until a line is read consisting only of *eofstring*. (By appending a minus (-) to the input redirection symbol (<<), leading tabs are deleted from each line of the input document before the shell passes the line to *command*.)

The shell creates a temporary file containing the input document and performs variable and command substitution on its contents before passing it to the command. Pattern matching on filenames is performed on the arguments of command lines in command substitutions. In order to prohibit all substitutions, you may quote any character of *eofstring*:

```
command <<\eofstring
```

The in-line input document feature is especially useful for small amounts of input data, where it is more convenient to place the data in the shell procedure than to keep it in a separate file. For instance, you could enter:

```
cat <<- xx
      This message will be printed on the
      terminal with leading tabs removed.
xx
```

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

Input / Output Redirection Using File Descriptors

We mentioned above that a command occasionally directs output to some file associated with a file descriptor other than 1 or 2. In languages such as C, one can associate output with any file descriptor by using the `write` (S) system call (see the *Programmer's Reference*). The shell provides its own mechanism for creating an output file associated with a particular file descriptor. By entering:

```
fd1 >& fd2
```

where *fd1* and *fd2* are valid file descriptors, one can direct output that would normally be associated with file descriptor *fd1* to the file associated with *fd2*. The default value for *fd1* and *fd2* is 1. If, at run time, no file is associated with *fd2*, then the redirection is void. The most common use of this mechanism is that of directing standard error output to the same file as standard output. This is accomplished by entering:

```
command 2>&1
```

If you wanted to redirect both standard output and standard error output to the same file, you would enter:

```
command 1>file 2>&1
```

The order here is significant: first, file descriptor 1 is associated with *file*; then file descriptor 2 is associated with the same file as is currently associated with file descriptor 1. If the order of the redirections were reversed, standard error output would go to the terminal, and standard output would go to *file*, because at the time of the error output redirection, file descriptor 1 still would have been associated with the terminal.

Supporting Commands and Features

This mechanism can also be generalized to the redirection of standard input. You could enter:

```
fda <& fdb
```

to cause both file descriptors *fda* and *fdb* to be associated with the same input file. If *fda* or *fdb* is not specified, file descriptor 0 is assumed. Such input redirection is useful for a command that uses two or more input sources.

Conditional Substitution

Normally, the shell replaces occurrences of *\$variable* by the string value assigned to *variable*, if any. However, there exists a special notation to allow conditional substitution, dependent upon whether the variable is set or not null. By definition, a variable is set if it has ever been assigned a value. The value of a variable can be the null string, which may be assigned to a variable in anyone of the following ways:

```
A=  
bcd=""  
efg=""  
set "" ""
```

The first three examples assign null to each of the corresponding shell variables. The last example sets the first and second positional parameters to null. The following conditional expressions depend upon whether a variable is set and not null. Note that the meaning of braces in these expressions differs from their meaning when used in grouping shell commands. *Parameter* as used below refers to either a digit or a variable name.

`${variable:-string}`

If *variable* is set and is nonnull, then substitute the value *\$variable* in place of this expression. Otherwise, replace the expression with *string*. Note that the value of *variable* is *not* changed by the evaluation of this expression.

`${variable:=string}`

If *variable* is set and is nonnull, then substitute the value *\$variable* in place of this expression. Otherwise, set *variable* to *string*, and then substitute the value *\$variable* in place of this expres-

sion. Positional parameters may not be assigned values in this fashion.

`${variable:?string}`

If *variable* is set and is nonnull, then substitute the value of *variable* for the expression. Otherwise, print a message of the form

variable: string

and exit from the current shell. (If the shell is the login shell, it is not exited.) If *string* is omitted in this form, then the message

variable: parameter null or not set

is printed instead.

`${variable:+string}`

If *variable* is set and is nonnull, then substitute *string* for this expression. Otherwise, substitute the null string. Note that the value of *variable* is not altered by the evaluation of this expression.

These expressions may also be used without the colon. In this variation, the shell does not check whether the variable is null or not; it only checks whether the variable has ever been set.

The two examples below illustrate the use of this facility:

1. This example performs an explicit assignment to the PATH variable:

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

This says, if PATH has ever been set and is not null, then it keeps its current value; otherwise, set it to the string “:/bin:/usr/bin”.

2. This example automatically assigns the HOME variable a value:

```
cd ${HOME:='/usr/gas'}
```

If HOME is set, and is not null, then change directory to it. Otherwise set HOME to the given value and change directory to it.

Invocation Flags

There are five flags that may be specified on the command line when invoking the shell. These flags may not be turned on with the `set` command:

- i If this flag is specified, or if the shell's input and output are both attached to a terminal, the shell is *interactive*. In such a shell, INTERRUPT (signal 2) is caught and ignored, and TERMINATE (signal 15) and QUIT (signal 3) are ignored.
- s If this flag is specified or if no input/output redirection arguments are given, the shell reads commands from standard input. Shell output is written to file descriptor 2. All remaining arguments specify the positional parameters.
- c When this flag is turned on, the shell reads commands from the first string following the flag. Remaining arguments are ignored.
- t When this flag is on, a single command is read and executed, then the shell exits. This flag is not useful interactively, but is intended for use with C programs.
- r If this flag is present the shell is a restricted shell (see `rsh` (C)).

Effective and Efficient Shell Programming

This section outlines strategies for writing efficient shell procedures, ones that do not waste resources in accomplishing their purposes. The primary reason for choosing a shell procedure to perform a specific function is to achieve a desired result at a minimum human cost. Emphasis should always be placed on simplicity, clarity, and readability, but efficiency can also be gained through awareness of a few design strategies. In many cases, an effective redesign of an existing procedure improves its efficiency by reducing its size, and often increases its comprehensibility. In any case, you should not worry about optimizing shell procedures unless they are intolerably slow or are known to consume an inordinate amount of a system's resources.

The same kind of iteration cycle should be applied to shell procedures as to other programs: write code, measure it, and optimize only the *few* important parts. The user should become familiar with the `time` command, which can be used to measure both entire procedures and parts thereof. Its use is strongly recommended; human intuition is notoriously unreliable when used to estimate timings of programs, even when the style of programming is a familiar one. Each timing test should be run several times, because the results are easily disturbed by variations in system load.

Number of Processes Generated

When large numbers of short commands are executed, the actual execution time of the commands may well be dominated by the overhead of creating processes. The procedures that incur significant amounts of such overhead are those that perform much looping, and those that generate command sequences to be interpreted by another shell.

If you are worried about efficiency, it is important to know which commands are currently built into the shell, and which are not. Here is the alphabetical list of those that are built in:

Effective and Efficient Shell Programming

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:

$$\text{processes} = (k * n) + c$$

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

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

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

```

:
#       split
trap 'rm temp$$; trap 0; exit' 0 1 2 3 15
start1=0 start2=0
b='[A-Za-z]'
```

```

cat > temp$$
        # read stdin into temp file
        # save original lengths of $1, $2
if test -s "$1"
then start1=`wc -l < $1`
fi
if test -s "$2"
then start2=`wc -l < $2`
fi
grep "$b" temp$$ >> $1
        # lines with letters onto $1
grep -v "$b" temp$$ | grep '[0-9]' >> $2
        # lines without letters onto $2
total=" `wc -l < temp$$` "
end1=" `wc -l < $1` "
end2=" `wc -l < $2` "
lost=" `expr $total - \($end1 - $start1\) \
- \($end2 - $start2\) ` "
echo "$total read, $lost thrown away"
```

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

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

7

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

Effective and Efficient Shell Programming

the order is irrelevant. For instance, the second of the following examples is likely to be faster because the input to `sort` will be much smaller:

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

Shortening Directory Searches

Directory searching can consume a great deal of time, especially in those applications that utilize deep directory structures and long pathnames. Judicious use of `cd`, the *change directory* command, can help shorten long pathnames and thus reduce the number of directory searches needed. As an exercise, try the following commands:

```
ls -l /usr/bin/* >/dev/null
cd /usr/bin; ls -l * >/dev/null
```

The second command will run faster because of the fewer directory searches.

Directory-Search Order and the PATH Variable

The `PATH` variable is a convenient mechanism for allowing organization and sharing of procedures. However, it must be used in a sensible fashion, or the result may be a great increase in system overhead.

The process of finding a command involves reading every directory included in every pathname that precedes the needed pathname in the current `PATH` variable. As an example, consider the effect of invoking `nrdf` (i.e., `/usr/bin/nrdf`) when the value of `PATH` is “`:/bin:/usr/bin`”. The sequence of directories read is:

```
.
/
/bin
/
/usr
/usr/bin
```

This is a total of six directories. A long path list assigned to `PATH` can increase this number significantly.

The vast majority of command executions are of commands found in */bin* and, to a somewhat lesser extent, in */usr/bin*. Careless *PATH* setup may lead to a great deal of unnecessary searching. The following four examples are ordered from worst to best with respect to the efficiency of command searches:

```
:/usr/john/bin:/usr/localbin:/bin:/usr/bin
:/bin:/usr/john/bin:/usr/localbin:/usr/bin
:/bin:/usr/bin:/usr/john/bin:/usr/localbin
/bin:./usr/bin:./usr/john/bin:./usr/localbin
```

The first one above should be avoided. The others are acceptable and the choice among them is dictated by the rate of change in the set of commands kept in */bin* and */usr/bin*.

A procedure that is expensive because it invokes many short-lived commands may often be speeded up by setting the *PATH* variable inside the procedure so that the fewest possible directories are searched in an optimum order.

Good Ways to Set Up Directories

It is wise to avoid directories that are larger than necessary. You should be aware of several special sizes. A directory that contains entries for up to 30 files (plus the required *.* and *..*) fits in a single disk block and can be searched very efficiently. One that has up to 286 entries is still a small directory; anything larger is usually a disaster when used as a working directory. It is especially important to keep login directories small, preferably one block at most. Note that, as a rule, directories never shrink. This is very important to understand, because if your directory ever exceeds either the 30 or 286 thresholds, searches will be inefficient; furthermore, even if you delete files so that the number of files is less than either threshold, the system will still continue to treat the directory inefficiently.

Shell Procedure Examples

The power of the UNIX shell command language is most readily seen by examining how many labor-saving UNIX 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

```

Shell Procedure Examples

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 *User's Reference* if you are completely unfamiliar with these commands. The **tr** command translates all characters except letters and digits into newline characters, and then squeezes out repeated newline characters. This leaves each string (in this case, any contiguous sequence of letters and digits) on a separate line. The **sort** command sorts the lines and emits only one line from any sequence of one or more repeated lines. The next **tr** converts everything to lowercase, so that identifiers differing only in case become identical. The output is sorted again to bring such duplicates together. The “**uniq -d**” prints (once) only those lines that occur more than once, yielding the desired list.

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

```
cmd1 | cmd2 | cmd3

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

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

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

DRAFT

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

Users often write this kind of procedure for convenience in dealing with commands that require the use of distinct flags that cannot be given default values that are reasonable for all (or even most) users.

EDFIND

```
:
# Usage: edfind file arg
# Finds the last occurrence in "file" of a line
# whose beginning matches "arg", then prints
# 3 lines (the one before, the line itself,
# and the one after)
ed - $1 << -EOF
    ?^$2?
    -,+p
    q
EOF
```

This illustrates the practice of using `ed` in-line input scripts into which the shell can substitute the values of variables.

Shell Procedure Examples

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

```

7

The *mkfiles* procedure uses output redirection to create zero-length files. The `expr` command is used for counting iterations of the `while` loop.

Shell Procedure Examples

NULL

```
:
# Usage: null files
# Create each of the named files as an empty file.
for eachfile
do
    >$eachfile
done
```

This procedure uses the fact that output redirection creates the (empty) output file if a file does not already exist.

PHONE

```
:
# Usage: phone initials ...
# Prints the phone numbers of the
# people with the given initials.
echo `initsext home`
grep "$1" <<END
    jfk 1234 999-2345
    lbj 2234 583-2245
    hst 3342 988-1010
    jqa 4567 555-1234
END
```

This procedure is an example of using an in-line input script to maintain a small database.

TEXTFILE

```

:
if test "$1" = "-s"
then
#   Return condition code
  shift
  if test -z "`$0 $*" " # check return value
  then
    exit 1
  else
    exit 0
  fi
fi

if test $# -lt 1
then echo "$0: Usage: $0 [ -s ] file ..." 1>&2
  exit 0
fi

file $* | fgrep ' text' | sed 's/:.*//'
```

To determine which files in a directory contain only textual information, *textfile* filters argument lists to other commands. For example, the following command line will print all the text files in the current directory:

```
pr `textfile *` | lpr
```

This procedure also uses an `-s` flag which silently tests whether any of the files in the argument list is a text file.

WRITEMAIL

```

:
#   Usage: writemail message user
#   If user is logged in,
#   writes message to terminal;
#   otherwise, mails it to user.
echo "$1" | { write "$2" || mail "$2" ;}
```

This procedure illustrates the use of command grouping. The message specified by `$1` is piped to both the `write` command and, if `write` fails, to the `mail` command.

Shell Grammar

item: *word*
 input-output
 name = value

simple-command: *item*
 simple-command item

command: *simple-command*
 (*command-list*)
 {*command-list*}
 for *name do command-list done*
 for *name in word do command-list done*
 while *command-list do command-list done*
 until *command-list do command-list done*
 case *word in case-part esac*
 if *command-list then command-list else-part fi*

pipeline: *command*
 pipeline | command

andor: *pipeline*
 andor && pipeline
 andor || pipeline

command-list: *andor*
 command-list ;
 command-list &
 command-list ; andor
 command-list & andor

input-output: > *file*
 < *file*
 << *word*
 >> *file*
 digit > file
 digit < file
 digit >> file

file: *word*
 & *digit*
 & -

case-part: *pattern*) *command-list* ;;

pattern: *word*
 pattern | *word*

else-part: **elif** *command-list* **then** *command-list* *else-part*
 else *command-list*
 empty

empty:

word: *a sequence of nonblank characters*

name: *a sequence of letters, digits, or underscores*
 starting with a letter

digit: **0 1 2 3 4 5 6 7 8 9**

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

Shell Grammar

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 (<code>`</code>)
"..."	Quote enclosed characters excepting: <code>\$`\"</code>

5. Reserved words

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

Chapter 8

The C-Shell

Introduction 8-1

Invoking the C-shell 8-2

Using Shell Variables 8-4

Using the C-Shell History List 8-7

Using Aliases 8-10

Redirecting Input and Output 8-12

Creating Background and Foreground Jobs 8-13

Using Built-In Commands 8-14

Creating Command Scripts 8-17

Using the argv Variable 8-18

Substituting Shell Variables 8-19

Using Expressions 8-21

Using the C-Shell: A Sample Script 8-22

Using Other Control Structures 8-25

Supplying Input to Commands 8-26

Catching Interrupts 8-27

Using Other Features 8-28

Starting a Loop at a Terminal 8-29

Using Braces with Arguments 8-31

Substituting Commands 8-32

Special Characters 8-33

Introduction

The C-shell program, `csh`, is a command language interpreter. The C-shell, like the standard UNIX shell `sh`, is an interface between you and the UNIX 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 *User's Reference*.

Invoking the C-shell

You can invoke the C-shell from another shell by using the `cs` command. To invoke the C-shell, enter:

```
cs
```

at the standard shell's command line. You can also direct the system to invoke the C-shell for you when you log in. If you have given the C-shell as your login shell in your `/etc/passwd` file entry, the system automatically starts the shell when you log in.

After the system starts the C-shell, the shell searches your home directory for the command files `.cshrc` and `.login`. If the shell finds the files, it executes the commands contained in them, then displays the C-shell prompt.

The `.cshrc` file typically contains the commands you wish to execute each time you start a C-shell, and the `.login` file contains the commands you wish to execute after logging in to the system. For example, the following is the contents of a typical `.login` file:

```
set ignoreeof
set mail=(/usr/spool/mail/bill)
set time=15
set history=10
mail
```

This file contains several `set` commands. The `set` command is executed directly by the C-shell; there is no corresponding UNIX 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 UNIX `mail` program is invoked.

When the C-shell finishes processing the *.login* file, it begins reading commands from the terminal, prompting for each with:

```
%
```

When you log out (by giving the **logout** command) the C-shell prints:

```
logout
```

and executes commands from the file *.logout* if it exists in your home directory. After that, the C-shell terminates and logs you off the system.

Using Shell Variables

The C-shell maintains a set of variables. For example, in the above discussion, the variables “history” and “time” had the values 10 and 15. Each C-shell variable has as its value an array of zero or more strings. C-shell variables may be assigned values by the `set` command, which has several forms, the most useful of which is:

```
set name = value
```

C-shell variables may be used to store values that are to be used later in commands through a substitution mechanism. The C-shell variables most commonly referenced are, however, those that the C-shell itself refers to. By changing the values of these variables you can directly affect the behavior of the C-shell.

One of the most important variables is “path”. This variable contains a list of directory names. When you enter a command name at your terminal, the C-shell examines each named directory in turn, until it finds an executable file whose name corresponds to the name you entered. The `set` command with no arguments displays the values of all variables currently defined in the C-shell.

The following example file shows typical default values:

```
argv ()
home /usr/bill
path (. /bin /usr/bin)
prompt %
shell /bin/csh
status 0
```

This output indicates that the variable “path” begins with the current directory indicated by dot (`.`), then `/bin`, and `/usr/bin`. Your own local commands may be in the current directory. Normal UNIX 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,

Using Shell Variables

you may accidentally overwrite a file which is valuable. If you prefer that the C-shell not overwrite files in this way you can:

```
set noclobber
```

in your *.login* file. Then entering:

```
date > now
```

causes an error message if the file *now* already exists. You can enter:

```
date >! now
```

if you really want to overwrite the contents of *now*. The “>!” is a special syntax indicating that overwriting or “clobbering” the file is ok. (The space between the exclamation point (!) and the word “now” is critical here, as “!now” would be an invocation of the history mechanism, described below, and have a totally different effect.)

Using the C-Shell History List

The C-shell can maintain a history list into which it places the text of previous commands. It is possible to use a notation that reuses commands, or words from commands, in forming new commands. This mechanism can be used to repeat previous commands or to correct minor typing mistakes in commands.

The following figure gives a sample session involving typical usage of the history mechanism of the C-shell. Boldface indicates user input:

Using the C-Shell History List

```
% cat bug.c
main()
{
    printf("hello);
}
% cc !$
cc bug.c
bug.c(4) :error 1: newline in constant
% ed !$
ed bug.c
28
3s/);/"/&/p
    printf("hello");
w
29
q
% !c
cc bug.c
% a.out
hello% !e
ed bug.c
29
3s/lo/lo\\n/p
    printf("hello\n");
w
31
q
% !c -o bug
cc bug.c -o bug
% size a.out bug
a.out: 5124 + 614 + 1254 = 6692 = 0x1b50
bug: 5124 + 616 + 1252 = 6692 = 0x1b50
% ls -l !*
ls -l a.out bug
-rwxr-xr-x 1 bill 7648 Dec 19 09:41 a.out
-rwxr-xr-x 1 bill 7650 Dec 19 09:42 bug
% bug
hello
% pr bug.c | lpt
lpt: Command not found.
% ^lpt^lpr
pr bug.c | lpr
%
```

Figure 8-1 Sample History Session

In this example, we have a very simple C program that has a bug or two in the file *bug.c*, which we **cat** out on our terminal. We then try to run the C compiler on it, referring to the file again as “!\$”, meaning the last argument to the previous command. Here the exclamation mark (!) is the history mechanism invocation metacharacter, and the dollar sign (\$) stands for the last argument, by analogy to the dollar sign in the editor which stands for the end-of-line.

The C-shell echoed the command, as it would have been typed without use of the history mechanism, and then executed the command. The compilation yielded error diagnostics, so we now edit the file we were trying to compile, fix the bug, and run the C compiler again, this time referring to this command simply as “!c”, which repeats the last command that started with the letter “c”.

If there were other commands beginning with the letter “c” executed recently, we could have said “!cc” or even “!cc;p” which prints the last command starting with “cc” without executing it, so that you can check to see whether you really want to execute a given command.

After this recompilation, we ran the resulting *a.out* file, and then noting that there still was a bug, ran the editor again. After fixing the program we ran the C compiler again, but tacked onto the command an extra “-o bug” telling the compiler to place the resultant binary in the file *bug* rather than *a.out*. In general, the history mechanisms may be used anywhere in the formation of new commands, and other characters may be placed before and after the substituted commands.

We then ran the *size* command to see how large the binary program images we have created were, and then we ran an “!s -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 “!pt”. 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 User's Reference*.

Using Aliases

The C-shell has an alias mechanism that can be used to make transformations on commands immediately after they are input. This mechanism can be used to simplify the commands you enter, to supply default arguments to commands, or to perform transformations on commands and their arguments. The alias facility is similar to a macro facility. Some of the features obtained by aliasing can be obtained also using C-shell command files, but these take place in another instance of the C-shell and cannot directly affect the current C-shell's environment or involve commands such as `cd` which must be done in the current C-shell.

For example, suppose there is a new version of the mail program on the system called *newmail* that you wish to use instead of the standard mail program *mail*. If you place the C-shell command

```
alias mail newmail
```

in your *.cshrc* file, the C-shell will transform an input line of the form:

```
mail bill
```

into a call on *newmail*. Suppose you wish the command `ls` to always show sizes of files, that is, to always use the `-s` option. In this case, you can use the `alias` command to do:

```
alias ls ls -s
```

or even:

```
alias dir ls -s
```

creating a new command named `dir`. If we then enter:

```
dir ~bill
```

the C-shell translates this to:

```
ls -s /usr/bill
```

Note that the tilde (`~`) is a special C-shell symbol that represents the user's home directory.

Thus the `alias` command can be used to provide short names for commands, to provide default arguments, and to define new short commands in terms of

other commands. It is also possible to define aliases that contain multiple commands or pipelines, showing where the arguments to the original command are to be substituted using the facilities of the history mechanism.

Thus the definition:

```
alias cd `cd \* ; ls`
```

specifies an `ls` command after each `cd` command. We enclosed the entire alias definition in single quotation marks (```) to prevent most substitutions from occurring and to prevent the semicolon (`;`) from being recognized as a metacharacter. The exclamation mark (`!`) is escaped with a backslash (`\`) to prevent it from being interpreted when the alias command is entered. The `"*"` here substitutes the entire argument list to the prealiasing `cd` command; no error is given if there are no arguments. The semicolon separating commands is used here to indicate that one command is to be done and then the next. Similarly the following example defines a command that looks up its first argument in the password file.

```
alias whois `grep \^ /etc/passwd`
```

The C-shell currently reads the `.cshrc` file each time it starts up. If you place a large number of aliases there, C-shells will tend to start slowly. You should try to limit the number of aliases you have to a reasonable number (10 or 15 is reasonable). Too many aliases causes delays and makes the system seem sluggish when you execute commands from within an editor or other programs.

Redirecting Input and Output

In addition to the standard output, commands also have a diagnostic output that is normally directed to the terminal even when the standard output is redirected to a file or a pipe. It is occasionally useful to direct the diagnostic output along with the standard output. For instance, if you want to redirect the output of a long running command into a file and wish to have a record of any error diagnostic it produces you can enter:

```
command > & file
```

The “> &” here tells the C-shell to route both the diagnostic output and the standard output into *file*. Similarly you can give the command:

```
command | & lpr
```

to route both standard and diagnostic output through the pipe to the 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.

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 **INTERUPT** or **QUIT** signals.

The **kill** command terminates a background job immediately. Normally, this is done by specifying the process number of the job you want killed. Process numbers can be found with the **ps** command.

Using Built-In Commands

This section explains how to use some of the built-in C-shell commands.

The **alias** command described above is used to assign new aliases and to display existing aliases. If given no arguments, **alias** prints the list of current aliases. It may also be given one argument, such as to show the current alias for a given string of characters. For example:

```
alias ls
```

prints the current alias for the string “ls”.

The **history** command displays the contents of the history list. The numbers given with the history events can be used to reference previous events that are difficult to reference contextually. There is also a C-shell variable named “prompt”. By placing an exclamation point (!) in its value the C-shell will substitute the number of the current command in the history list. You can use this number to refer to a command in a history substitution. For example, you could enter:

```
set prompt=^! % ^
```

Note that the exclamation mark (!) had to be escaped here even within back quotes.

The **logout** command is used to terminate a login C-shell that has “ignoreeof” set.

The **rehash** command causes the C-shell to recompute a table of command locations. This is necessary if you add a command to a directory in the current C-shell’s search path and want the C-shell to find it, since otherwise the hashing algorithm may tell the C-shell that the command wasn’t in that directory when the hash table was computed.

The **repeat** command is used to repeat a command several times. Thus to make 5 copies of the file *one* in the file *five* you could enter:

```
repeat 5 cat one >> five
```

The `setenv` command can be used to set variables in the environment. Thus:

```
setenv TERM adm3a
```

sets the value of the environment variable “TERM” to “adm3a”. The program `env` exists to print out the environment. For example, its output might look like this:

```
HOME=/usr/bill
SHELL=/bin/csh
PATH=/usr/ucb:/bin:/usr/bin:/usr/local
TERM=adm3a
USER=bill
```

The `source` command is used to force the current C-shell to read commands from a file. Thus:

```
source .cshrc
```

can be used after editing in a change to the `.cshrc` file that you wish to take effect before the next time you login.

The same holds true when using the `source` command with the `.login` file. The `time` command is used to cause a command to be timed no matter how much CPU time it takes. Thus:

```
time cp /etc/termcap /usr/bill/termcap
```

displays:

```
0.0u 0.4s 0:02 21%
```

Similarly:

```
time wc /etc/termcap /usr/bill/termcap
```

displays:

```
2071 5849 92890 /etc/termcap
2071 5849 92890 /usr/bill/termcap
4142 11698 185780 total
1.3u 0.7s 0:04 47%
```

This indicates that the `cp` command used a negligible amount of user time (u) and about 4/10ths of a second of system time (s); the elapsed time was 2

Using Built-In Commands

seconds (0:02). The word count command `wc` used 1.3 seconds of user time and 0.7 seconds of system time in 4 seconds of elapsed time. The percentage “47%” indicates that over the period when it was active the `wc` command used an average of 47 percent of the available CPU cycles of the machine.

The `unalias` and `unset` commands are used to remove aliases and variable definitions from the C-shell.

Creating Command Scripts

It is possible to place commands in files and to cause C-shells to be invoked to read and execute commands from these files, which are called C-shell scripts. This section describes the C-shell features that are useful when creating C-shell scripts.

Using the argv Variable

A `cs`h command script may be interpreted by saying:

```
cs script argument ...
```

where *script* is the name of the file containing a group of C-shell commands and *argument* is a sequence of command arguments. The C-shell places these arguments in the variable “`argv`” and then begins to read commands from *script*. These parameters are then available through the same mechanisms that are used to reference any other C-shell variables.

If you make the file *script* executable by doing:

```
chmod 755 script
```

or:

```
chmod +x script
```

and then place a C-shell comment at the beginning of the C-shell script (i.e., begin the file with a number sign (#)) then `/bin/csh` will automatically be invoked to execute *script* when you enter:

```
script
```

If the file does not begin with a number sign (#) then the standard shell `/bin/sh` will be used to execute it.

Substituting Shell Variables

After each input line is broken into words and history substitutions are done on it, the input line is parsed into distinct commands. Before each command is executed a mechanism known as variable substitution is performed on these words. Keyed by the dollar sign (\$), this substitution replaces the names of variables by their values. Thus:

```
echo $argv
```

when placed in a command script would cause the current value of the variable “argv” to be echoed to the output of the C-shell script. It is an error for “argv” to be unset at this point.

A number of notations are provided for accessing components and attributes of variables. The notation:

```
$?name
```

expands to 1 if *name* is set or to 0 if *name* is not set. It is the fundamental mechanism used for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The notation:

```
 $#name
```

expands to the number of elements in the variable “name”. To illustrate, examine the following terminal session (input is in boldface):

```
% set argv=(a b c)
% echo $?argv
1
% echo $#argv
3
% unset argv
% echo $?argv
0
% echo $argv
Undefined variable: argv.
%
```

It is also possible to access the components of a variable that has several values. Thus:

Substituting Shell Variables

`$argv[1]`

gives the first component of “argv” or in the example above “a”. Similarly:

`$argv[$#argv]`

would give “c”. Other notations useful in C-shell scripts are:

`$n`

where *n* is an integer. This is shorthand for:

`$argv[n]`

the *n*'th parameter and:

`$*`

which is a shorthand for:

`$argv`

The form:

`$$`

expands to the process number of the current C-shell. Since this process number is unique in the system, it is often used in the generation of unique temporary filenames.

One minor difference between “`$n`” and “`$argv[n]`” should be noted here. The form: “`$argv[n]`” will yield an error if *n* is not in the range 1-`$#argv` while “`$n`” will never yield an out-of-range subscript error. This is for compatibility with the way older shells handle parameters.

Another important point is that it is never an error to give a subrange of the form: “`n-`”; if there are less than “*n*” components of the given variable then no words are substituted. A range of the form: “`m-n`” likewise returns an empty vector without giving an error when “*m*” exceeds the number of elements of the given variable, provided the subscript “*n*” is in range.

Using Expressions

To construct useful C-shell scripts, the C-shell must be able to evaluate expressions based on the values of variables. In fact, all the arithmetic operations of the C language are available in the C-shell with the same precedence that they have in C. In particular, the operations “==” and “!=” compare strings and the operators “&&” and “|” implement the logical AND and OR operations.

The C-shell also allows file inquiries of the form:

```
-? filename
```

where question mark (?) is replaced by a number of single characters. For example, the expression primitive:

```
-e filename
```

tells whether *filename* exists. Other primitives test for read, write and execute access to the file, whether it is a directory, or if it has nonzero length.

It is possible to test whether a command terminates normally, by using a primitive of the form:

```
{ command }
```

which returns 1 if the command exits normally with exit status 0, or 0 if the command terminates abnormally or with exit status nonzero. If more detailed information about the execution status of a command is required, it can be executed and the “status” variable examined in the next command. Since “\$status” is set by every command, its value is always changing.

For the full list of expression components, see `csH(C)` in the *User's Reference*.

Using the C-Shell: A Sample Script

A sample C-shell script follows that uses the expression mechanism of the C-shell and some of its control structures:

```
#
# Copyc copies those C programs in the specified list
# to the directory ~/backup if they differ from the files
# already in ~/backup
#
set noglob
foreach i ($argv)

    if ($i != *.c) continue # not a .c file so do nothing

    if (! -r ~/backup/$i:t) then
        echo $i:t not in backup... not cp\`ed
        continue
    endif

    cp -s $i ~/backup/$i:t # to set $status

    if ($status != 0) then
        echo new backup of $i
        cp $i ~/backup/$i:t
    endif
end
```

This script uses the **foreach** command, which iteratively executes the group of commands between the **foreach** and the matching **end** statements for each value of the variable “*i*”. If you want to look more closely at what happens during execution of a **foreach** loop, you can use the debug command **break** to stop execution at any point and the debug command **continue** to resume execution. The value of the iteration variable (*i* in this case) will stay at whatever it was when the last **foreach** loop was completed.

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.

Using the C-Shell: A Sample Script

The more general **if** statements above also admit a sequence of **else-if** pairs followed by a single **else** and an **endif**, for example:

```
if ( expression ) then
    commands
else if ( expression ) then
    commands
...
else
    commands
endif
```

Another important mechanism used in C-shell scripts is the colon (:) modifier. We can use the modifier **:r** here to extract the root of a filename or **:e** to extract the extension. Thus if the variable **"i"** has the value */mnt/foo.bar* then

```
echo $i $i:r $i:e
```

produces:

```
/mnt/foo.bar /mnt/foo bar
```

This example shows how the **:r** modifier strips off the trailing **".bar"** and the **:e** modifier leaves only the **"bar"**. Other modifiers take off the last component of a pathname leaving the head **:h** or all but the last component of a pathname leaving the tail **:t**. These modifiers are fully described in the **csH(C)** page in the *User's Reference*. It is also possible to use the command substitution mechanism to perform modifications on strings to then reenter the C-shell environment. Since each usage of this mechanism involves the creation of a new process, it is much more expensive to use than the colon (:) modification mechanism. It is also important to note that the current implementation of the C-shell limits the number of colon modifiers on a **"\$"** substitution to 1. Thus:

```
%echo $i $i:h:t
```

produces:

```
/a/b/c /a/b:t
```

and does not do what you might expect.

Finally, we note that the number sign character (**#**) lexically introduces a C-shell comment in C-shell scripts (but not from the terminal). All subsequent characters on the input line after a number sign are discarded by the C-shell. This character can be quoted using **"#"** or **"\#"** to place it in an argument word.

Using Other Control Structures

The C-shell also has control structures **while** and **switch** similar to those of C. These take the forms:

```
while ( expression )
    commands
end
```

and:

```
switch ( word )

case str1:
    commands
    breaksw

...

case strn:
    commands
    breaksw

default:
    commands
    breaksw

endsw
```

For details see the manual section for **csh(C)**. C programmers should note that we use **breaksw** to exit from a **switch** while **break** exits a **while** or **foreach** loop. A common mistake to make in C-shell scripts is to use **break** rather than **breaksw** in switches.

Finally, the C-shell allows a **goto** statement, with labels looking like they do in C:

```
loop:
    commands
    goto loop
```

Supplying Input to Commands

Commands run from C-shell scripts receive by default the standard input of the C-shell which is running the script. It allows C-shell scripts to fully participate in pipelines, but mandates extra notation for commands that are to take inline data.

Thus we need a metanotation for supplying inline data to commands in C-shell scripts. For example, consider this script which runs the editor to delete leading blanks from the lines in each argument file:

```
# deblank -- remove leading blanks
foreach i ($argv)
ed - $i << ' EOF'
1,$s/^[ ]*//
w
q
'EOF'
end
```

The notation:

```
<< 'EOF'
```

means that the standard input for the `ed` command is to come from the text in the C-shell script file up to the next line consisting of exactly EOF. The fact that the EOF is enclosed in single quotation marks (`'`), i.e., it is quoted, causes the C-shell to not perform variable substitution on the intervening lines. In general, if any part of the word following the `<<` which the C-shell uses to terminate the text to be given to the command is quoted then these substitutions will not be performed. In this case since we used the form `'1,$'` in our editor script we needed to insure that this dollar sign was not variable substituted. We could also have insured this by preceding the dollar sign (`$`) with a backslash (`\`), i.e.:

```
1,\<$s/^[ ]*//
```

Quoting the EOF terminator is a more reliable way of achieving the same thing.

Catching Interrupts

If our C-shell script creates temporary files, we may wish to catch interruptions of the C-shell script so that we can clean up these files. We can then do:

```
onintr label
```

where *label* is a label in our program. If an interrupt is received the C-shell will do a “goto label” and we can remove the temporary files, then do an `exit` command (which is built in to the C-shell) to exit from the C-shell script. If we wish to exit with nonzero status we can write:

```
exit (1)
```

to exit with status 1.

Using Other Features

There are other features of the C-shell useful to writers of C-shell procedures. The **verbose** and **echo** options and the related **-v** and **-x** command line options can be used to help trace the actions of the C-shell. The **-n** option causes the C-shell only to read commands and not to execute them and may sometimes be of use.

One other thing to note is that the C-shell will not execute C-shell scripts that do not begin with the number sign character (**#**), that is C-shell scripts that do not begin with a comment.

There is also another quotation mechanism using the double quotation mark (**"**), which allows only some of the expansion mechanisms we have so far discussed to occur on the quoted string and serves to make this string into a single word as the single quote (**'**) does.

Starting a Loop at a Terminal

It is occasionally useful to use the **foreach** control structure at the terminal to aid in performing a number of similar commands. For instance, if there were three shells in use on a particular system, */bin/sh*, */bin/nsh*, and */bin/csh*, you could count the number of persons using each shell by using the following commands:

```
grep -c csh$ /etc/passwd
grep -c nsh$ /etc/passwd
grep -c -v /sh$ /etc/passwd
```

Because these commands are very similar we can use **foreach** to simplify them:

```
$ foreach i ('sh$' 'csh$' '-v sh$')
? grep -c $i /etc/passwd
? end
```

Note here that the C-shell prompts for input with “?” when reading the body of the loop. This occurs only when the **foreach** command is entered interactively.

Also useful with loops are variables that contain lists of filenames or other words. For example, examine the following terminal session:

```
% set a=(`ls`)
% echo $a
csh.n csh.rm
% ls
csh.n
csh.rm
% echo $#a
2
```

The **set** command here gave the variable “a” a list of all the filenames in the current directory as value. We can then iterate over these names to perform any chosen function.

The output of a command within back quotation marks (``) is converted by the C-shell to a list of words. You can also place the quoted string within double quotation marks (") to take each (nonempty) line as a component of the variable. This prevents the lines from being split into words

Starting a Loop at a Terminal

at blanks and tabs. A modifier `:x` exists which can be used later to expand each component of the variable into another variable by splitting the original variable into separate words at embedded blanks and tabs.

Using Braces with Arguments

Another form of filename expansion involves the characters, “{” and “}”. These characters specify that the contained strings, separated by commas (,) are to be consecutively substituted into the containing characters and the results expanded left to right. Thus:

```
A{str1,str2,...strn}B
```

expands to:

```
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,...}
```

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 *User’s Reference* for more information.

Special Characters

The following table lists the `cs`h and UNIX special characters. A number of these characters also have special meaning in expressions. See the `cs`h 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

Special Characters

Input/output metacharacters

- < Indicates redirected input
- > Indicates redirected output

Expansion/Substitution Metacharacters

- \$ Indicates variable substitution
- ! Indicates history substitution
- : Precedes substitution modifiers
- ^ Used in special forms of history substitution
- ` Indicates command substitution

Other Metacharacters

- # Begins scratch filenames; indicates C-shell comments
- Prefixes option (flag) arguments to commands

Chapter 9

The Korn Shell

Introduction 9-1

Starting ksh 9-2

Using the ksh Built-in Editors 9-3

 Using the vi Built-In Editor Modes 9-4

 Editing in Input Mode 9-4

 Editing in Control Mode 9-5

Accessing Commands in the History File 9-8

 Displaying Commands in the History File 9-8

 Reexecuting Previous Commands 9-8

 Editing Previous Commands 9-9

Using Job Control 9-10

 Referring to Jobs 9-10

 Using the ksh Job Control Commands 9-11

 Running Jobs in the Background 9-11

 Moving Background Jobs to Foreground 9-12

 Moving Foreground Jobs to Background 9-12

 Displaying Information about Jobs 9-13

 Terminating a Background Job 9-13

Customizing the ksh Environment 9-14

 Modifying the .profile File 9-14

 Executing a File on Logout 9-15

 Modifying the .kshrc File 9-15

 Defining Aliases 9-15

 Setting ksh Options 9-16

 Modifying the ksh History File 9-17

Manipulating Commands Wider Than the Screen 9-19

Using Expanded cd Capabilities 9-20

Introduction

The Korn Shell, **ksh(C)**, is an interactive command-language interpreter and programming language that reads and executes commands from either the terminal or a file. The **ksh** combines the best features of the two common UNIX System shells, the standard Bourne shell, **sh(C)**, and the C shell, **csh(C)**. The **ksh** provides both compatibility with **sh** and the command history and substitution features of **csh**. In addition, **ksh** includes command-line editing, job control, and enhanced command-history functionality.

Starting ksh

If you are currently running **sh** or **csk**, you can start **ksh** by entering **ksh** at the command line. However, when you run **ksh** as a program from your original login shell, you cannot access the command-line editing and job control features. To take advantage of these features, you must run **ksh** as your login shell.

To use **ksh** as your default login shell, ask the system administrator to change your login-shell specifier in the */etc/passwd* file to **ksh**. When the system administrator specifies **ksh** as the login shell when creating a new user, the **sysadmsh(ADM)** utility creates two files in the user's home directory: *.profile* and *.kshrc*.

When you log in using **ksh** as your login shell, the shell reads commands from the system profile file, */etc/profile*, and then from *.profile* in the current directory or *\$HOME/.profile*, if either file exists. Next, the shell reads commands from the **ksh** environment file, *\$HOME/.kshrc*, if it exists. If there is no *.sh_history* file (the history file where **ksh** stores commands that you enter at the keyboard) in the your home directory, **ksh** creates one.

For more information on these files and how to modify your **ksh** environment, see the section later in this chapter, "Customizing the ksh Environment."

Using the ksh Built-in Editors

Using **cs**h or **sh**, the only way to fix errors on the command line is to backspace or retype the entire line. With **ksh**, you can edit the command line using the familiar commands that you use to edit files. The **ksh** provides both **vi**-like and **emacs**-like built-in editor interfaces for editing the command line.

At login time, **ksh** reads the *.kshrc* environment file and turns on the **vi**-like editor. You can turn off the **ksh** editor functionality completely or turn off **vi** and turn on **emacs** for the current session or for each login session.

To turn off **vi** for the current login session only, enter the following at the command line:

```
set +o vi
```

To turn on **emacs** for the current login session, enter the following at the command line:

```
set -o emacs
```

To turn on or off either the **vi** or **emacs** editors automatically when you log in, add the appropriate command to the *.profile* or environment file (*.kshrc* by default).

You can also use the **EDITOR** and **VISUAL** environment variables to set the editor to any pathname that ends in **vi**. For example, to turn on the **vi** editor automatically when you log in, add the following line to your *.kshrc* file:

```
EDITOR=/usr/bin/vi
```

Note

The **VISUAL** variable overrides the **EDITOR** variable.

This chapter includes information on the built-in **vi**-like editor. For information about using the **emacs**-like editor, see the **ksh(C)** reference page, under the section “Emacs Editing Mode.” Or consult your **emacs**

Using the ksh Built-in Editors

documentation, if you have installed **emacs** separately.

Using the vi Built-In Editor Modes

Like the **vi** text editor, **ksh**'s built-in **vi** editor has two modes: *input mode* and *control mode*. In input mode, **ksh** inserts the characters that you type at the keyboard in an editing buffer. In control mode, **ksh** interprets the characters that you enter at the keyboard as editing commands.

When you log in using **ksh** as your login shell, you are in input mode automatically. (This differs from the **vi** text editor; you are initially in control mode and you must press **a** or **i** to begin entering text.) To enter control mode from input mode, press **<ESC>**. If you press **<ESC>** while in control mode, the terminal beeps.

Editing in Input Mode

While entering commands in input mode, you can edit the command line using editing commands from the following table:

Input Mode Editing Commands

Command	Description
<CTL>h or <BKSP>	moves back one character
<Return> or <CTL>m	executes the current line
<CTL>v	escapes the character that follows (for entering control characters)

Editing in Control Mode

At any time before you press `<Return>` to execute the command, you can press `<ESC>` to enter control mode. In control mode, you can move around the command line as if you were in `vi`, editing a file.

The following table shows the `vi` commands for moving the cursor on the command line in control mode:

Moving the Cursor

Key	Description
<code>h</code>	moves left one character
<code>l</code>	moves right one character
<code>b</code>	moves left one word
<code>B</code>	moves left one word, skipping punctuation
<code>w</code>	moves right one word
<code>W</code>	moves right one word, skipping punctuation
<code>e</code>	moves to the last character of the next word
<code>E</code>	moves to the last character of the next word, skipping punctuation
<code>0</code>	moves to the beginning of the current line
<code>\$</code>	moves to the end of the current line
<code>^</code>	moves to the first character on the current line that is not a <code><Space></code> or <code><TAB></code>
<code>fx</code>	moves right to the next occurrence of <code>x</code>
<code>Fx</code>	moves left to the preceding occurrence of <code>x</code>
<code>tx</code>	moves right to the character before the next occurrence of <code>x</code>
<code>Tx</code>	moves left to the character following the preceding occurrence of <code>x</code>
<code>;</code>	repeats the last character search <code>f</code> , <code>F</code> , <code>t</code> , or <code>T</code> .
<code>,</code>	reverses the last character search <code>f</code> , <code>F</code> , <code>t</code> , or <code>T</code> .

Using the ksh Built-in Editors

The following table gives the commands for entering input mode from control mode, and for changing and deleting text:

Adding, Changing, and Deleting Text	
Key	Description
a	enters input mode after the character under the cursor
A	enters input mode after the last character on the line
i	enters input mode before the character under the cursor
I	enters input mode before the first character on the line
_	appends the last word of the previous ksh command to the current line and then enters input mode.
rz	replaces the character under the cursor with z
Rtext	replaces characters with <i>text</i> beginning at the cursor
cmotion	changes the characters from cursor position, using the <i>vi motion</i> command For example:
cw	changes word below cursor
cl	changes character below cursor and then adds text
c\$	changes from the current character to the end of line
cc	deletes the entire line and returns to input mode (same as c\$)
x	deletes the character under the cursor
X	deletes the character to the left of the cursor
dw	Deletes the word under cursor
dmotion	deletes characters, starting at the cursor, up to and including the other end of <i>motion</i>
D	deletes from the cursor to the end of line
d\$	same as D
dd	deletes the entire line
ymotion	yanks the current character using <i>vi motion</i> command
Y	yanks from cursor to end of line
y\$	same as Y
yy	yanks the entire line into the buffer
p	puts previously yanked (or deleted) words to the right of the cursor
P	puts previously yanked (or deleted) words to the left of the cursor

The following table shows the control mode commands for executing and redrawing the current line, repeating commands, and undoing modifications on the command line:

Miscellaneous Control Mode Commands

Command	Description
<code><Return></code> or <code><CTL>m</code>	executes the current line
<code><CTL>l</code>	redraws the current line
<code>~</code>	changes the case of the character under the cursor
<code>.</code>	repeats the most recent <code>vi</code> command
<code>u</code>	undoes the previous <code>vi</code> command
<code>U</code>	undoes all modifications on the current line

Accessing Commands in the History File

Using the `vi` built-in editor, you can access previously entered commands that are stored in your `ksh` history file (`.sh history` by default). Once you retrieve a command, you can modify it and execute it again.

Displaying Commands in the History File

To display the list of the commands that are stored in the history file, enter `history`. The `history` command is a predefined alias that uses the `ksh` built-in command, `fc` (fix command), to access the history file. For more information about `fc`, see `ksh(C)`.

The `history` alias displays the last 16 (or fewer, if there are fewer than 16 commands in the file) commands in the history file. You can specify how many and which commands that you want `history` to display. Note that the commands must be accessible in the history file for `history` to display them. The following list gives examples of how to use the `history` alias:

- `history -4` displays the previous four commands only.
- `history 20` displays all commands from the history file, starting with 20.
- `history 12 24` displays only commands 12 through 24.

Reexecuting Previous Commands

The `ksh` also includes `r`, another predefined alias that uses the `fc` built-in command. The `r` alias allows you to reexecute commands from the history file. This alias functions similarly to the `!` command in `csh`. The following list shows some common uses of the `r` alias:

Alias	Description
<code>r</code>	reexecutes the last command entered
<code>r command</code>	reexecutes the last <i>command</i> entered

r *x* reexecutes the last command beginning with *x*
r # reexecutes command number #

Note that **r** simply reexecutes commands from the history list; **r** does not allow you to modify commands before you execute them.

Editing Previous Commands

You can use **vi** commands to search for and retrieve commands from the history file. Once you locate a command, you can edit and reexecute it using the **vi** commands described in the section “Editing in Control Mode” earlier in this chapter.

To move through the history file, first press **(ESC)** to enter control mode. Then, use the **vi** commands in the following table to move up and down in the history file:

Moving in the History File	
Command	Description
k	moves up (previous) one command in the history file
j	moves down (next) one command in history file
/string	searches left and up (back) through the history file for the next command containing <i>string</i>
?string	searches right and down (forward) through the history file for the next command containing <i>string</i>
G	goes back to the oldest accessible command in the history file
n	repeats the last / or ? search command
N	repeats the last / or ? command, searching backward

Using Job Control

The job control feature of **ksh** allows you manipulate jobs running in the foreground and background. With job control, you can stop and restart programs, and move them between the foreground and the background.

Like **sh** and **csk**, the **ksh** runs commands in the foreground by default; the shell waits for the current command to finish executing before displaying the prompt. You run a command in the background by adding an ampersand (&) character to the end of the command before pressing <Return>. Before displaying the prompt, the shell displays the status of any completed background jobs.

When job control is active and you run a command in the background, **ksh** displays both the number of the job in square brackets, [], and the process ID number (PID).

Job control is active by default. If job control is disabled, **ksh** displays the following error message when you try to manipulate foreground and background jobs:

```
no job control
```

To activate job control, add the following line to your environment file (*.kshrc*):

```
set -o monitor
```

To use job control, Suspend must be set (usually <CTL>z). To set it, add the following line to your \$HOME/*profile* file:

```
stty susp '^Z'
```

Referring to Jobs

When you use **ksh** as your login shell, you can refer to jobs in several different ways:

<i>PID</i>	refers to the job by the process ID number.
<i>%number</i>	refers to the job by the job number that <i>ksh</i> displays in square brackets.
<i>%string</i>	refers to the job whose command begins with <i>string</i> .
<i>%?string</i>	refers to the job whose command contains <i>string</i> .
<i>%+ or %%</i>	refers to the current job.
<i>%-</i>	refers to the previous job.

For example, if you enter the command `sleep 30&`, and *ksh* displays:

```
[1] 3456
```

you can refer to this background job in various ways, including the following:

```
3456
%1
%sle
%?ee
```

Using the *ksh* Job Control Commands

The *ksh* includes the following built-in commands for process control. These commands take a PID, job name, or number as the argument.

Job Control Commands

Command	Description
<code>bg</code>	starts the specified stopped job in the background.
<code>fg</code>	moves the specified background job to the foreground.
<code>jobs</code>	displays status information about current jobs.
<code>kill</code>	terminates the specified background job.
<code>wait</code>	tells <i>ksh</i> to wait for all or a specific background process to complete.

Running Jobs in the Background

To run a job in the background, enter the command name, followed by an ampersand (&) character and press <Return>.

For example, when you enter `sleep 30&`, *ksh* starts the job in the background and displays a message like the following:

Using Job Control

```
[1] 3456
```

When the job finishes, **ksh** displays a message like this:

```
[1] + Done          sleep 30&
```

Moving Background Jobs to Foreground

To move job that is running in the background to the foreground, enter **fg** followed by the PID, command name, or job number.

For example, move the **sleep** background process to the foreground by entering:

```
fg %?ee
```

Once the job is in the foreground, **ksh** waits for it to complete before displaying a prompt.

Moving Foreground Jobs to Background

To move a foreground job to the background, press Suspend. The **ksh** stops the job and displays a message like the following:

```
[1] + Stopped      sleep 30&
```

The **ksh** displays your prompt. At the prompt, enter **bg** followed by the PID, job number, or name. For example, move the **sleep** process to the background by entering:

```
bg %?ee
```

The **ksh** restarts the suspended job, displays a message like the following, and runs the job in the background:

```
[1] sleep 30&
```

Displaying Information about Jobs

Use the **jobs** command to display status information about the jobs that **ksh** is currently running. To display information about all active jobs, enter **jobs** at the command line. The **ksh** displays status information in the following format:

```
[2] + Running          sleep 40&
[1] - Stopped         sleep 30&
```

The **ksh** displays a plus (+) character after the job number of the current job, a minus (-) after the previous job.

Use the **-l** option to display the PID after the job number. For example, if you enter **jobs -l**, **ksh** displays information like the following:

```
[1] + 23909 Stopped      sleep 30&
```

You can limit the display to PID only by using the **-p** option to **jobs**.

Terminating a Background Job

To terminate a process, use the **kill(C)** command. The syntax for **kill** is:

```
kill [-signal] job
```

where *signal* is an optional signal number or name and *job* is the job name, number, or PID of the job that you want to terminate.

To display a complete list of signal numbers and names, enter **kill -l**. If you do not specify *signal*, **ksh** sends the TERM (terminate) signal to the specified job. See the previous section “Referring to Jobs” in this chapter for more information on referring to jobs by job name, number, and PID.

When you use **kill**, **ksh** displays a message like the following and terminates the job:

```
[1] + Terminated      sleep 30&
```

Customizing the ksh Environment

The **ksh** uses two files located in your home directory, *.profile* and *.kshrc*, to set up your environment. Use the *.profile* file to set variables and options for your login shell and other programs that you run from the **ksh**. The *.kshrc* file is the **ksh**-specific environment file; use it to define aliases and set **ksh** command-line options.

Modifying the .profile File

Whenever you log in using **ksh** as your login shell, **ksh** executes the *.profile* file, automatically executing commands and setting exported environment variables. Commands and environment variables in *.profile* must be in the same format as they are when you enter them from the keyboard.

To assign values to **ksh** environment variables, use this format:

```
EDITOR=/bin/vi
export EDITOR
```

The following table shows some of the more common environment variables:

Environment Variables	
Variable	Description
COLUMNS	specifies the number of columns that ksh uses to display the command line
EDITOR	sets either the vi -like or emacs -like editor to use when editing the command line
ENV	sets the environment file (<i>.kshrc</i> by default)
HISTFILE	sets an alternate history file (<i>.sh_history</i> by default)
HISTSIZE	sets the maximum number of commands that are stored in the history file (128 by default)
HOME	specifies the default argument used by the cd(C) command
MAILCHECK	specifies the interval in seconds that ksh checks for new mail (600 seconds by default)

PATH	specifies the pathnames that ksh searches when executing commands
PS1	specifies the primary prompt to display when the interactive option is on (\$ by default); ksh replaces an exclamation point (!) with the command number (to print a ! in the prompt, enter !!)
TERM	specifies the type of terminal that you are using
VISUAL	sets the editor to use when editing command lines (overrides the value of EDITOR)

For example, to set up your prompt to include the machine name and command number, add the following following lines to your *.profile*:

```
PS1='! fscott'
export PS1
```

For a complete list of environment variables used by **ksh**, see **ksh(C)**.

Executing a File on Logout

You can use the **trap 0** command in your *.profile* file to instruct **ksh** to execute a file, for example, *logout*, when you exit the shell. To do this, enter the following in *.profile*:

```
trap $HOME/.logout 0
```

The *logout* file must be executable.

Modifying the .kshrc File

The *kshrc* file contains definitions for aliases and functions and default option settings for **ksh**. You should specify any commands and definitions that only **ksh** recognizes in this file rather than in *.profile*.

Defining Aliases

An alias is an abbreviated name for a command. Use the following format to define an alias:

```
alias shortname='commandname'
```

where *shortname* is the abbreviated name for *commandname*. For example, to define **ls** as an alias for **lc -F**, use the following command:

```
alias ls='lc -F'
```

Customizing the ksh Environment

You can define aliases for the current **ksh** session by entering **alias** at the command line. To specify that an alias definition remain in effect across login sessions, add the alias command to your *.kshrc* (or other **ksh** environment file).

You can display the complete list of your aliases by entering **alias** at the prompt. To display a particular alias definition, enter **alias** followed by the alias name. For example, if you enter **alias ls**, **ksh** displays:

```
ls=lc -F
```

To unset a particular alias (in either the *.kshrc* file or at the command line), enter **unalias** followed by the alias name.

When you run scripts that do not invoke another **ksh**, regular aliases do not remain defined. However, you can *export* these alias definitions by defining them in your environment file using this format:

```
alias -x ls='lc -F'
```

The **ksh** automatically predefines several aliases. These aliases are compiled into the shell, but you can unset or redefine them. (We do not recommend redefining preset aliases.) The two most common preset aliases are **history** (for displaying the contents of the *.sh_history* file) and **r** (for reexecuting previously entered commands). For more information about preset aliases, see the section “Accessing Commands in the History File” in this chapter and in **ksh(C)**.

Setting ksh Options

You can set **ksh** command-line options in your environment file. Use the following format:

```
set -o option
```

To unset an option, use a plus (+) character in place of minus (-).

The following table lists some useful **ksh** options:

set Options	
Option	Description
allexport	exports all subsequent variables automatically (same as -a)
bgnice	runs all background jobs at a lower priority (set by default)
emacs	uses the emacs -like built-in editor for command-line editing
ignoreeof	prevents ksh from exiting on end-of-file, <CTL>d ; (when ignoreeof is set, you must enter exit to terminate the shell)
monitor	runs background jobs as a separate process group and prints status when a process completes (systems with job control set the monitor option automatically for interactive shells)
verbose	prints shell input lines as they are read
vi	uses the vi -like built-in editor for command-line editing

For a complete list of **set** options and descriptions, see **ksh(C)**.

Modifying the ksh History File

The **ksh** stores the commands that you enter at the keyboard in the *.sh_history* file in your home directory by default. You can specify a different history file using the **HISTFILE** environment variable in your *.profile* file.

For example, to use the file *history* instead of *.sh_history*, add the following lines to your *.profile*:

```
HISTFILE=~/.history
export HISTFILE
```

You can specify the maximum number of previously entered commands that you can retrieve from the history file with the **HISTSIZE** environment variable. If **HISTSIZE** is not set, **ksh** stores 128 commands by default. There is no limit to the number of commands that the **ksh** can store.

Note

If **HISTSIZE** is very large, **ksh** may be very slow at startup time.

Customizing the ksh Environment

The **ksh** does not delete the history file when you exit; the shell appends and stores commands across login sessions. When you log in, **ksh** deletes any commands in your history file that are older than the last number of commands specified by **HISTSIZE**.

Manipulating Commands Wider Than the Screen

The **ksh** allows you to enter commands of up to 256 characters from the terminal. You can define the maximum width of the command-line display (80 columns by default) using the **COLUMNS** variable. (See the section “Customizing the **ksh** Environment” in this chapter for more information about setting environment variables.)

If you enter or edit a command that is wider than the command-line display minus two columns, **ksh** automatically scrolls the command line horizontally to the left or right of your screen. In the last column on the right side of the screen, **ksh** displays one of the following characters to show that the line is scrolling:

- < scrolls to the right (text to the left is not displayed)
- > scrolls to the left (text to the right is not displayed)
- * text both to the right and to the left is not displayed

For example, if **COLUMNS** is not set (the width of the command-line display is the default of 80 columns) and your command line is greater than 78 characters wide, **ksh** scrolls the command line left to display the end of the line. To the right of the command line, **ksh** displays the > character.

Using Expanded cd Capabilities

The **ksh** includes expanded functionality for the **cd(C)** command. You can instruct **ksh** to search through a specified list of directories when you enter pathnames that do not begin with the slash (/) character. To do this, set the **CDPATH** variable in your *.profile* file. (See the section “Customizing the ksh Environment” for more information about setting environment variables.)

The **ksh** provides an option to **cd** that allows you to return quickly to your previous working directory. For example, if you are in the */usr/spool/mail* directory and you enter:

```
cd /usr/bin
```

you can return to your previous directory, */usr/spool/mail*, by entering **cd -**. From this directory, you can enter **cd -** again to return to the */usr/spool/mail* directory.

The **ksh** provides a means for changing to a directory with a pathname that is slightly different from your current working directory. To do this, use the following format:

```
cd old new
```

where *old* is the part of the pathname that you want to change and *new* is what you want to change it to. For example, if you are in */usr/spool/mail* and you want to change to */usr/bin/mail*, enter:

```
cd spool bin
```

Chapter 10

Using A Trusted System

- Introduction 10-1
 - Terminology 10-1
 - The Security Administrator 10-2
- Login Security 10-3
 - Logging In 10-3
 - What To Do If You Cannot Log In 10-3
 - Changing Your Password 10-4
 - Using Another Account 10-6
- Using Commands On A Trusted System 10-7
 - Authorizations 10-7
 - The auths(C) Command 10-8
 - Using Promains 10-10
 - Security For Files In Sticky Directories 10-11
 - Commands You Cannot Use 10-11
- Recommended Security Practices 10-12
 - Password Security 10-12
 - Logging In And Out 10-13
 - File Security 10-13
 - Running Untrustworthy Programs 10-14
- Data Encryption—Commands and Descriptions 10-16
 - crypt—Encode/Decode Files 10-17
 - Encrypting and Decrypting With Editors 10-18



Introduction

Every computer system needs protection from people accessing the computer, disks and system files without the system administrator's permission. The operating system carries its own protection in the form of built-in security features not present in other UNIX systems. These features apply to all users of the system and are maintained by the system administrator.

This chapter describes security from the viewpoint of the ordinary user. If you find that your system does not show a feature discussed in this chapter, your administrator has switched it off.

This chapter includes the following:

- Terminology used to describe ways of enforcing and breaking security.
- The role of the security administrator.
- How to log into a trusted system, change password and use another account.
- How to issue commands on a trusted system.
- Recommended security practices and security tips.

Terminology

The following terms are used to describe ways of enforcing and breaking security:

A *Trojan Horse* is a program which masquerades as an innocent program. It allows a person to steal your data, corrupt your files or gain access to your account.

A *login spoofing program* disguises itself as the login program in order to steal your password. The program displays the login prompt on the terminal and waits for you to type in your user name followed by password. When you respond to the prompts and enter your user name and password, the program stores the password and reports that your entry was incorrect. The spoofing program then ends and the correct login program starts.

Introduction

A *protected subsystem* is a collection of files, devices and commands which protects a set of resources or which performs security tasks.

The *trusted computing base* (TCB) of a system is the software, hardware and firmware which provide the system with security. The TCB of your equipment consists of the system hardware, the firmware, and the operating system.

The Security Administrator

A security administrator is appointed to enforce security practices, monitor the system, trace attempts to breach security and return the system to a trusted state in the unlikely event of a security break-in.

Login Security

This section describes how to log into a trusted system, change password and use another account. It also explains what to do if you have difficulty logging in.

Logging In

The following login prompts are displayed:

```
login: user name
Password: non-echoed password
```

When you enter your password correctly, the last times you successfully and unsuccessfully logged in are displayed:

```
Last successful login for user: date and time
Last unsuccessful login for user: date and time
```

If these times do not match your actions, consult your administrator immediately. Someone may have tried to log into your account.

What To Do If You Cannot Log In

If you cannot log in, go through the following checklist:

- The security administrator has given your password a lifetime, which has now expired. Ask the administrator to change your password and re-open your account.
- The security administrator has set a limit to the number of unsuccessful login attempts you are allowed to make for your account. When you exceed this number your account is locked automatically. Ask the administrator to re-open the account. If you feel that you entered your login details correctly, tell the administrator immediately. It is possible that the system has been interfered with.

Login Security

- The security administrator has set a limit to the number of unsuccessful login attempts allowed at your terminal. When this number is exceeded your account is locked automatically. Ask the administrator to re-open the account. If you believe that you entered your login details correctly, inform the administrator immediately.
- The security administrator has locked your account or terminal. To continue work you must ask the administrator to re-open the account.
- The security administrator has set a date by which your password expires. When your password expires you are prompted to change it.
- If you forget your password, ask the security administrator to change it.

Changing Your Password

The security administrator decides whether or not you can change password for yourself. The administrator can also set a minimum time period between changes of password.

If You Are Not Allowed To Change Password For Yourself

If you are not allowed to change password for yourself and you try to use the `passwd(C)` command, the message below is displayed. You must ask the administrator to change your password.

```
Password cannot be changed. Reason: Not allowed to
execute password for the given user.
```

If You Are Allowed To Change Password For Yourself

If you are allowed to change the password for yourself, the administrator sets up your account to allow you to specify the password of your choice or have the system generate one for you.

When you use the `passwd(C)` command you are prompted for your current password:

Old password:

When you type it in correctly the date and time of your last change of password are displayed:

Last successful password change for *user*: *date and time*
Last unsuccessful password change for *user*: *date and time*

Make sure that these messages reflect your last attempts to change password. If they do not, tell your administrator immediately.

The following prompt is then displayed:

Choose password

You can choose whether you pick your own password, or have the system create one for you.

1. Pick your own password
2. Pronounceable password will be generated for you

Enter choice (default is 1):

If you enter **1**, you are prompted for your new password. You are then prompted to repeat your entry. Refer to the “Recommended Security Practices” section for guidelines on choosing a password.

If you select **2**, the system generates a password for you. The following message is displayed:

Generating random pronounceable password for *user*.
The password, along with the hyphenated version, is shown.
Hit <RETURN> or <ENTER> until you like the choice.
When you have chosen the password you want, type it in.
Note: Type your interrupt character or 'quit' to abort at any time.

Password:xxxxx Hyphenation:xx-xx-xx Enter password:

The generated password is displayed with a hyphenated version. The hyphenation separates the password into logical parts and is designed to help you commit the password to memory. Do not write the password down.

If you decide not to change your password type **quit** or your interrupt character (normally the key). Your last unsuccessful password change time is updated and the following message is displayed:

Login Security

Password cannot be changed. Reason: user stopped program.

Using Another Account

The `su(C)` command allows you to use another account. The security administrator may impose restrictions on the way you can use the `su(C)` command. For example, you may not be allowed to `su` to administrator accounts or the root account.

Using Commands On A Trusted System

The use of commands is restricted on a trusted system. You can issue certain commands only if the security administrator has given you the appropriate *authorization*. This section describes the different types of *authorization* and how they affect your use of commands.

Authorizations

The security mechanism has two types of authorization: kernel and subsystem. A kernel authorization allows you to run specific processes on the operating system. A subsystem authorization allows you to use the commands of a specific protected subsystem.

The kernel authorizations are as follows:

execsuid	This authorization allows you to run SUID programs. An SUID program gains access to all the files, processes and resources belonging to the person running the program or the owner of the program file.
nopromain	defines SUID program behavior. If the nopromain authorization is on, SUID programs run as on traditional UNIX systems. Otherwise, a <i>promain</i> , for <i>protected domain</i> , is created, in which programs are less likely to be able to corrupt or steal your private data.
chmodsugid	Allows you to change the <code>setuid</code> and <code>setgid</code> attributes of a file or directory, using the <code>chmod(C)</code> command.
chown	Allows you to change the ownership of files using the <code>chown(C)</code> command.

There are two levels of subsystem authorization: primary and secondary. A primary subsystem authorization allows you to use the commands of a protected subsystem as an administrator. Primary authorizations are given to administrators and are fully described in the Administrator's Guide. However, the authorizations below can be given to ordinary users:

Using Commands On A Trusted System

mem	This authorization allows you to use the ps(C) command to check the status of other users' processes, and the ipcs(C) command to report the status of inter-process communication. Without the authorization you can only use these commands to report on processes belonging to you.
terminal	Allows you to use the write(C) command to communicate with other users. If you use the write(C) command without the authorization, any control codes and escape sequences in your message are converted to ASCII characters.

A secondary subsystem authorization allows you to use the commands of a subsystem as an ordinary user i.e. you are not given administrative privilege. Secondary authorizations are described below:

printqueue	Allows you to view other users' jobs on the print queue.
printerstat	Allows you to use the enable(C) and disable(C) commands to change the status of printers.
queryspace	Allows you to use the df(C) command to query the amount of space available on the file systems.

Note that you cannot gain the subsystem authorizations of another account by changing to the account using the **su(C)** command. Also, you do not drop any authorizations which the new account does not possess.

The **auths(C)** Command

The **auths(C)** command allows you to list your kernel authorizations, and start up a shell so that you can issue commands with specific authorizations. The instructions below show you how to use the **auths(C)** command with a variety of arguments. Examples are given for a user with **execsuid** and **chown** authorizations.

- The **auths(C)** command without arguments lists your kernel authorizations. For example:

```
$ auths
Kernel authorizations: execsuid, chown
```

- The **auths(C)** command with the **-a** option allows you to specify a restricted set of one or more of your authorizations. For example, the user with **execsuid** and **chown** authorizations can restrict himself/herself to the **chown** authorization:

```
$ auths -a chown
$ auths
Kernel authorizations: chown
```

To restore your authorizations, leave the shell started by the **auths(C) -a** command.

- The **auths(C)** command with the **-r** option allows you to specify which of your authorizations you wish to remove. For example:

```
$ auths -r chown
$ auths
Kernel authorizations: execsuid
```

You must leave the shell started by the **auths(C) -r** command to restore your authorizations.

- The **auths(C)** command with the **-c** option allows you to issue a command instead of starting an interactive subshell. In the example below, **chown** authorization is removed and then the **auths(C)** command is issued. The result is a line listing the user's authorizations; the **chown** authorization is not included.

```
$ auths -r chown -c auths
Kernel authorizations: execsuid
```

When the user lists his/her authorizations, the **chown** authorization is restored:

```
$ auths
Kernel authorizations: execsuid, chown
```

Using Promains

The `promain` feature allows you to control the damage a SUID program can do to your files. Recall that a SUID program starts execution with *effective* user ID equal to the *owner* of the SUID program file and *real* user ID equal to the *invoker* of the SUID program. On traditional UNIX systems, a SUID program has complete access to all files, processes, and IPC objects (collectively called *resources*) to which the invoker *or* the owner has access, because the program can use `setuid(S)` to switch between the invoker and owner user ID. Outside a `promain`, this power is restricted to resources to which the invoker *and* the owner have access, as described in this section.

On UNIX systems, the SUID feature is used when one user (or system function) needs to protect files from access except through a well-defined set of programs. An example is the suite of line printer commands, which work with a set of configuration files, status files, and shell scripts to keep track of which print jobs are queued to which printers. Users and line printer administrators use several commands to submit and cancel jobs, change and query the status of printers, and add and remove printers from the system or from active duty. All printer files are owned by the pseudo user `lp`, the user ID which is the owner of all files used by the line printer subsystem, including the printer special devices themselves.

When you invoke the `lp(C)` command to print a file, the program can access the files in the database, but can also access files that you request to be printed because the program can `setuid` to your user ID to access your files. A malicious `lp` program could just as easily look for protected files in your directory hierarchy and copy them to a place protected such that only `lp` could read them. Thus, the fact that you trust `lp` enough to run it as a program means that you trust that it will not abuse the power you give when you run it.

If you run a SUID program in the `and` and the `nopromain` kernel authorization is `off`, a `promain` is created, and the current directory is noted as the *promain root*. Files in the subtree starting at the `promain root` are said to be *inside* the `promain`, while files outside that subtree are said to be *outside* the `promain`. `Promains` protect a user against a malicious SUID program by restricting the kinds of accesses the program can do outside the `promain` when running as you (the invoker). When running with the invoker's effective user ID outside the `promain`, the program can access files if *both* the invoker and the owner have access (public files). Inside the `promain`, the program has access according to the normal rules. Refer to the `promain(M)` page in the *User's Reference* for examples of how to use `promains`.

Security For Files In Sticky Directories

You can remove a file from a directory which has its sticky bit set, only if you own the file.

Commands You Cannot Use

A trusted system prevents you from using networking commands. In order to transport information, the security administrator must record information on disk or tape.

Recommended Security Practices

This section gives a list of recommended security practices for the ordinary user.

Password Security

It is your responsibility to protect your password. The careless use and maintenance of passwords represents the greatest threat to the security of a computer system. Here are some guidelines for choosing and maintaining passwords:

- A password should be at least eight characters in length and include letters, digits and punctuation marks. For example, **frAiJ6***.
- Do not use a password that is easy to guess. A password must not be a name, nickname, proper noun or word found in the dictionary. Do not use your birthdate or a number in your address.
- Do not use words spelled backwards.
- Do not start or end a password with a single number. For example, do not use **terry9** as your password.
- Use different passwords on different machines. Do not make the passwords reflect the names of the machines.
- Always keep your password secret. A password should never be written down, shared with another person, sent over electronic mail or spoken.
- Never re-use a password. This just increases the probability of someone guessing it.
- Never type a password while someone is watching your fingers.

Logging In And Out

The following guidelines include “things to look out for” as well as recommended practices for logging into your system and logging out.

- When logging into a trusted system, check that the reported last login and logout times are as you remember them. Look out for login attempts made when you are normally logged out of the system. Report any discrepancies to your security administrator, immediately.
- Be careful how you type in your password.
- When you enter your password and the system reports an error, although you believe your entry to have been correct, tell your security administrator immediately. Check the reported last login time against the current time. If there is a discrepancy it is possible that a spoofing program (see “Terminology” section) has taken your password.
- Never leave yourself logged in at an unattended terminal.

File Security

Follow the guidelines below when you are creating, copying and moving files. The list also includes security tips related to your startup scripts.

- When you create a file or directory your startup script determines the permissions given to the file or directory. Newly created files and directories should only be accessible by you (the owner) or the group. It is advisable to keep your files and directories secure in this way, by leaving your startup script set to the system default. If you wish to share a few files with other users, change the permissions on those files, individually.
- When you use the `cp(C)` command to copy an SUID file owned by someone else, the new file is also an SUID file and is owned by you. Note that when an SUID file is executed, it has access to all your files and directories. It is good practice to use the `chmod(C)` command to change the permissions on the file so that it can be accessed only by you.
- When you use the `cp(C)` command to copy a file so as to create a new file, the new file takes the permissions of the original file. Remember to check the permissions of the new file and, if necessary, change them using the `chmod(C)` command.

Recommended Security Practices

- Remember that temporary directories are world-readable.
- Use the `ls(C)` command to check the permissions on your shell, mailer and startup files. If the files can be read and modified by other users, change the permissions using the `chmod(C)` command so that only you have access to them.

Running Untrustworthy Programs

This section gives instructions for running an untrustworthy program. The instructions show you how to protect all files, which you do not want a program to access, by protecting private files outside a restricted directory.

A restricted sub-hierarchy can be created by making a *gateway* directory in your directory hierarchy. This acts as a *ceiling* for programs running in directories underneath the gateway. It prevents malicious programs from opening files using relative pathnames (those not beginning with a `/`), and prevents programs from searching, opening or deleting files using absolute pathnames.

The instructions are as follows:

1. Remove `execsuid`, `chmodsuid` and `chown` authorizations from your program using the `auths(C)` command. This stops the program from running other SUID programs, from creating SUID files with your ownership and from giving your files away with the `chown(C)` command:

```
auths -r execsuid,chmodsuid,chown
```

2. Create a gateway directory. Then move all files to be manipulated by the program into a subdirectory of the gateway directory. Then change the gateway directory's mode to `000` using the `chmod(C)` command:

```
mkdir /usr/you/gateway
mkdir /usr/you/gateway/test
cd /usr/you/gateway/test
cp /usr/you/testfiles/* .
chmod 000 /usr/you/gateway
ls -ldu /usr/you/gateway
```

Note the time reported by the `ls(C)` command, as it is the time that the `chmod(C)` command was performed on the gateway directory. The time should be unchanged unless the program was malicious.

3. Change your home directory's permission to no access:

```
chmod 000 /usr/you
```

4. Note the *change* time associated with your home directory:

```
cd  
ls -ldu
```

You can discover if the program has opened and closed your directory, by using the inode change time for the directory.

5. Start the program. Relative pathnames starting at */usr/you/gateway/test* are stopped at the gateway directory and any other attempts to access files in your directory hierarchy are stopped at your home directory.
6. Check for background processes. A malicious program can start a malicious process which waits to do its damage until later.
7. After running the program, look at any error messages and hidden files (using the `ls(C)` command with `-a` option). Note especially the change times on your home directory and on your gateway directory:

```
ls -ldu /usr/you  
chmod 750 /usr/you  
ls -ldu /usr/you/gateway
```

8. If you wish, restore permissions on your gateway directory so that it can be removed:

```
chmod 750 /usr/you/gateway
```

You may now copy out the results of the program, remove the gateway hierarchy and so on.

This procedure protects most of your directory hierarchy from harm. However, sophisticated attacks, which change permissions on the gateway and home directories and destroy the integrity of your files, cannot be discovered until after the event. If you suspect the program, back up your directory hierarchy before running the program, or analyze the program's source code before using it.

Data Encryption—Commands and Descriptions

If you have sensitive data that requires greater protection than that provided by access permission, you can encrypt the data. The encrypted file can not be read without a password. If somebody tries to read the encrypted file without a password, it cannot be understood.

Note

You will only have data encryption capabilities if the **crypt(C)** software is installed on your system. This software is available only within the United States and must be requested from your distributor.

There are seven different commands used in data encryption. A brief summary of these commands appears in the following table.

COMMAND LINE	DESCRIPTION
crypt	This command is used to encode and decode files. The crypt command reads from the standard input or keyboard and writes to the standard output or terminal.
makekey	This command generates an encryption key.
ed -x	This command line edits a file that has already been encrypted, or creates a new encrypted file using the ed editor.
vi -x	This command line edits a file that has already been encrypted, or creates a new encrypted file using the vi editor.

ex -x	This command line edits a file that has already been encrypted, or creates a new encrypted file using the ex editor.
edit -x	This command line edits a file that has already been encrypted, or creates a new encrypted file using the edit editor.
X	This command encrypts a file while in the editor mode (ed , ex , or edit).

crypt—Encode/Decode Files

The **crypt** command encodes and decodes files for security. When using **crypt**, you have to assign a password (key) to encode the file. The same password is used to decode the file. An encrypted file cannot be read unless the correct password is used to decode it.

If no password is given with the **crypt** command, the system will prompt you for one. For security, the screen does not display the password as you type it in.

Password security is the most vulnerable part of the **crypt** command. Anyone who figures out your password can look at your files. The best way to ensure your security is to select an uncommon group of characters. As with your login password, the password should be no more than eight letters or numbers long.

A file can be encrypted in the shell mode using **crypt**, or in the edit mode using the **-x** or **X** option. When you are ready to decrypt the file, you can use the **crypt** command in the shell mode. The following is the command format to encrypt a file:

```
crypt < oldfile > newfile
```

Before removing the unencrypted *oldfile*, make sure the encrypted *newfile* can be decrypted using the appropriate password. The *oldfile* is the file to be encrypted. The *newfile* is the name of the destination file for the encrypted text. The *oldfile* should now be removed. The system will prompt you for a password.

Note

Always remember to remove the file (*oldfile*) from which you are encrypting because it will not be encrypted. Only the *newfile* will be encrypted.

Without any arguments, the **crypt** command takes standard input from the keyboard and encodes it before directing it to the standard output (the display). To encode an existing file, you must tell **crypt** to take its input (<) from a file instead of the keyboard. Similarly, you must tell **crypt** to send its output (>) to a new file instead of the display.

To decrypt a file, redirect the encrypted file to a new file you can read. The command to decrypt a file is as follows:

```
crypt < crypted_file > new_filename
```

Note

Always encrypt and decrypt files separately.

Encrypting and Decrypting With Editors

The editors (**ed**, **edit**, **ex**, and **vi**) can be used to either edit an existing file that has been encrypted or to create a new encrypted file by using the **-x** option. When encrypting a file, you have to assign a password to encode the file. The same password is used to decode the file. An encrypted file cannot be read unless the correct password is used to decode it.

Select an uncommon group of characters for the password. It should be no more than eight characters long.

The following is the command format for the editors (**ed**, **edit**, **ex**, and **vi**) using the **-x** option:

```
ed -x [filename]
```

```
edit -x [filename]
```

```
ex -x [filename]
```

```
vi -x [filename]
```

The **-x** option is used either to edit an existing file that has been encrypted or to create a new encrypted file. The *filename* variable is the name of the file that is being created or edited. The system will prompt you for a password.

When you get ready to decrypt the file, you must use the **crypt** command from the shell.

The editor **X** command is another way to encrypt a file while in the editor mode. The **X** command will only work with the **ed**, **edit**, or **ex** editors. (For the **vi** editor, type **:X**.) This command also needs a password to encrypt and decrypt files.

After you have edited the file, you can easily encrypt it again by using the **X** command as follows:

1. While still in the editor, enter **X** on a line by itself.
2. The system will prompt you for a password.
3. Quit the file.

Chapter 11

Simple Programming with awk

Introduction 11-1

Basic awk 11-2

Program Structure 11-2

Usage 11-3

Fields 11-4

Printing 11-4

Formatted Printing 11-6

Simple Patterns 11-6

Simple Actions 11-8

A Handful of Useful One-liners 11-9

Error Messages 11-10

Patterns 11-11

BEGIN and END 11-11

Relational Expressions 11-12

Regular Expressions 11-13

Combinations of Patterns 11-16

Pattern Ranges 11-17

Actions 11-18

Built-in Variables 11-18

Arithmetic 11-18

Strings and String Functions 11-21

Field Variables 11-25

Number or String? 11-26

Control Flow Statements 11-27

Arrays 11-30

User-Defined Functions 11-32

Some Lexical Conventions 11-33

Output 11-34

The print Statement 11-34

Output Separators 11-34

The printf Statement 11-35

Output into Files 11-37

Output into Pipes 11-37

Input	11-39
Files and Pipes	11-39
Input Separators	11-39
Multi-Line Records	11-40
The getline Function	11-40
Command-Line Arguments	11-43
Using awk with Other Commands and the Shell	11-45
The system Function	11-45
Cooperation with the Shell	11-45
Example Applications	11-48
Generating Reports	11-48
Additional Examples	11-50
awk Summary	11-53
Command Line	11-53
Patterns	11-53
Control Flow Statements	11-53
Input-Output	11-54
Functions	11-54
String Functions	11-54
Arithmetic Functions	11-55
Operators (Increasing Precedence)	11-55
Regular Expressions (Increasing Precedence)	11-56
Built-in Variables	11-56
Limits	11-56
Initialization, Comparison, and Type Coercion	11-57

Introduction

Suppose you want to tabulate some survey results stored in a file, print various reports summarizing these results, generate form letters, reformat a data file for one application package to use with another package, or count the occurrences of a string in a file. **awk** is a programming language that makes it easy to handle these and many other tasks of information retrieval and data processing. The name **awk** is an acronym constructed from the initials of its developers; it denotes the language and also the Altos UNIX System V system command you use to run an **awk** program.

awk is an easy language to learn. It automatically does quite a few things that you have to program for yourself in other languages. As a result, many useful **awk** programs are only one or two lines long. Because **awk** programs are usually smaller than equivalent programs in other languages, and because they are interpreted, not compiled, **awk** is also a good language for prototyping.

The first part of this chapter introduces you to the basics of **awk** and is intended to make it easy for you to start writing and running your own **awk** programs. The rest of the chapter describes the complete language and is somewhat less tutorial. For the experienced **awk** user, there is a summary of the language at the end of the chapter.

You should be familiar with UNIX commands and shell programming to use this chapter. Although you do not need other programming experience, some knowledge of the C programming language is beneficial because many constructs found in **awk** are also found in C.

Basic awk

This section provides enough information for you to write and run some of your own programs. Each topic presented is discussed in more detail in later sections.

Program Structure

The basic operation of **awk**(C) is to scan a set of input lines one after another, searching for lines that match any set of patterns or conditions that you specify. For each pattern, you can specify an action; this action is performed on each line that matches the pattern. Accordingly, an **awk** program is a sequence of pattern-action statements, as Figure 11-1 shows.

Structure:

```
pattern { action }
pattern { action }
...
```

Example:

```
$1 == "address" { print $2, $3 }
```

Figure 11-1 awk Program Structure and Example

The example in the figure is a typical **awk** program, consisting of one pattern-action statement. The program prints the second and third fields of each input line whose first field is *address*. In general, **awk** programs work by matching each line of input against each of the patterns in turn. For each pattern that matches, the associated action (which may involve multiple steps) is executed. Then the next line is read, and the matching starts over. This process typically continues until all the input has been read.

Either the pattern or the action in a pattern-action statement may be omitted. If there is no action with a pattern, as in

```
$1 == "name"
```

the matching line is printed. If there is no pattern with an action, as in

```
{ print $1, $2 }
```

the action is performed for every input line. Because patterns and actions are both optional, actions are enclosed in braces to distinguish them from patterns.

Usage

There are two ways to run an **awk** program. First, you can type the command line to execute the pattern-action statements on the set of named input files:

```
awk 'pattern-action statements' optional list of input files
```

For example, you could say

```
awk '{ print $1, $2 }' file1 file2
```

Notice that the pattern-action statements are enclosed in single quotes. This protects characters like **\$** from being interpreted by the shell and also allows the program to be longer than one line.

If no files are mentioned on the command line, **awk(C)** reads from the standard input. You can also specify that input comes from the standard input by using the hyphen (**-**) as one of the input files. For example, to read input first from *file1* and then from the standard input, enter:

```
awk '{ print $3, $4 }' file1 -
```

The arrangement above is convenient when the **awk** program is short (a few lines). If the program is long, it is often more convenient to put it into a separate file and use the **-f** option to fetch it:

```
awk -f program file optional list of input files
```

For example, the following command line says to fetch and execute *myprogram* on input from the file *file1*:

```
awk -f myprogram file1
```

Basic awk

Fields

awk normally reads its input one line, or record, at a time; a record is, by default, a sequence of characters ending with a newline character. **awk** then splits each record into fields, where, by default, a field is a string of non-blank, non-tab characters.

As input for many of the **awk** programs in this chapter, we use the file *countries*, which contains information about the 10 largest countries in the world. Each record contains the name of a country, its area in thousands of square miles, its population in millions, and the continent on which it is found. (Data are from 1978; the U.S.S.R. has been arbitrarily placed in Asia.) The white space between fields is a tab in the original input; a single blank space separates both North and South from America.

USSR	8650	262	Asia
Canada	3852	24	North America
China	3692	866	Asia
USA	3615	219	North America
Brazil	3286	116	South America
Australia	2968	14	Australia
India	1269	637	Asia
Argentina	1072	26	South America
Sudan	968	19	Africa
Algeria	920	18	Africa

Figure 11-2 The Sample Input File *countries*

This file is typical of the kind of data **awk** is good at processing — a mixture of words and numbers separated into fields by blanks and tabs.

The number of fields in a record is determined by the field separator. Fields are normally separated by sequences of blanks and/or tabs, so the first record of *countries* would have four fields, the second five, and so on. It is possible to set the field separator to just tab, so each line would have four fields, matching the meaning of the data. We will show how to do this shortly. For the time being, let's use the default: fields separated by blanks and/or tabs. The first field within a line is called \$1, the second \$2, and so forth. The entire record is called \$0.

Printing

If the pattern in a pattern-action statement is omitted, the action is executed for all input lines. The simplest action is to print each line; you can accomplish this with an **awk** program consisting of a single **print** statement

```
{ print }
```

The command line

```
awk '{ print }' countries
```

prints each line of *countries*, copying the file to the standard output. The **print** statement can also be used to print parts of a record; for instance, this program prints the first and third fields of each record:

```
{ print $1, $3 }
```

Thus, entering

```
awk '{ print $1, $3 }' countries
```

produces as output the sequence of lines:

```
USSR 262  
Canada 24  
China 866  
USA 219  
Brazil 116  
Australia 14  
India 637  
Argentina 26  
Sudan 19  
Algeria 18
```

When printed, items separated by a comma in the **print** statement are separated by the output field separator, which, by default, is a single blank. Each line printed is terminated by the output record separator, which by default is a newline.

Note

In the remainder of this chapter, we only show **awk** programs, without the command line that invokes them. Each complete program can be run, either by enclosing it in quotes as the first argument of the **awk** command, or by putting it in a file and invoking **awk** with the **-f** flag, as discussed in “awk Command Usage.” In an example, if no input is mentioned, the input is assumed to be the file *countries*.

Formatted Printing

For more carefully formatted output, **awk** provides a C-like **printf** statement

```
printf format, expr1, expr2, ..., exprn
```

that prints the *expr_n*'s according to the specification in the string *format*. For example, the **awk** program

```
{ printf "%10s %6d\n", $1, $3 }
```

prints the first field (*\$1*) as a string of 10 characters (right justified), then a space, then the third field (*\$3*) as a decimal number in a six-character field, then a newline (*\n*). With input from the file *countries*, this program prints an aligned table:

USSR	262
Canada	24
China	866
USA	219
Brazil	116
Australia	14
India	637
Argentina	26
Sudan	19
Algeria	18

With **printf**, no output separators or newlines are produced automatically; you must create them yourself by using *\n* in the format specification. “The **printf** Statement” in this chapter contains a full description of **printf**.

Simple Patterns

You can select specific records for printing or other processing by using simple patterns. **awk** has three kinds of patterns. First, you can use patterns called relational expressions that make comparisons. For example, the operator **==** tests for equality. To print the lines for which the fourth field equals the string *Asia*, we can use the program consisting of the single pattern

```
$4 == "Asia"
```

With the file *countries* as input, this program yields

```
USSR      8650    262    Asia
China     3692    866    Asia
India     1269    637    Asia
```

The complete set of comparisons is `>`, `>=`, `<`, `<=`, `==` (equal to) and `!=` (not equal to). These comparisons can be used to test both numbers and strings. For example, suppose we want to print only countries with a population greater than 100 million. All that is needed is the program

```
$3 > 100
```

(Remember that the third field in the file *countries* is the population in millions.) It prints all lines in which the third field exceeds 100.

Second, you can use patterns called regular expressions that search for specified characters to select records. The simplest form of a regular expression is a string of characters enclosed in slashes:

```
/US/
```

This program prints each line that contains the (adjacent) letters US anywhere; with the file *countries* as input, it prints

```
USSR      8650    262    Asia
USA       3615    219    North America
```

We will have a lot more to say about regular expressions later in this chapter.

Third, you can use two special patterns, `BEGIN` and `END`, that match before the first record has been read and after the last record has been processed. This program uses `BEGIN` to print a title:

```
BEGIN { print "Countries of Asia:" }
/Asia/ { print "    ", $1 }
```

The output is

```
Countries of Asia:
    USSR
    China
    India
```

Simple Actions

We have already seen the simplest action of an **awk** program: printing each input line. Now let's consider how you can use built-in and user-defined variables and functions for other simple actions in a program.

Built-in Variables

Besides reading the input and splitting it into fields, **awk(C)** counts the number of records read and the number of fields within the current record; you can use these counts in your **awk** programs. The variable **NR** is the number of the current record, and **NF** is the number of fields in the record. So the program

```
{ print NR, NF }
```

prints the number of each line and how many fields it has, while

```
{ print NR, $0 }
```

prints each record preceded by its record number.

User-Defined Variables

Besides providing built-in variables like **NF** and **NR**, **awk** lets you define your own variables, which you can use for storing data, doing arithmetic, and the like. To illustrate, consider computing the total population and the average population represented by the data in the file *countries*.

```
    { sum = sum + $3 }  
END { print "Total population is", sum, "million"  
      print "Average population of", NR, "countries is", sum/NR }
```

Note

awk initializes *sum* to zero before it is used.

The first action accumulates the population from the third field; the second action, which is executed after the last input, prints the sum and average:

```
Total population is 2201 million  
Average population of 10 countries is 220.1
```

Functions

awk has built-in functions that handle common arithmetic and string operations for you. For example, there is an arithmetic function that computes square roots. There is also a string function that substitutes one string for another. **awk** also lets you define your own functions. Functions are described in detail in the section “Actions” in this chapter.

A Handful of Useful One-liners

Although **awk** can be used to write large programs of some complexity, many programs are not much more complicated than what we have seen so far. Here is a collection of other short programs that you may find useful and instructive. They are not explained here, but any new constructs do appear later in this chapter.

Print last field of each input line:

```
{ print $NF }
```

Print 10th input line:

```
NR ==
```

Print last input line:

```
END { line = $0 }
      { print line }
```

Print input lines that don't have four fields:

```
NF != 4 { print $0, "does not have 4 fields" }
```

Print input lines with more than four fields:

```
NF > 4
```

Print input lines with last field more than 4:

```
$NF > 4
```

Basic awk

Print total number of input lines:

```
END          { print NR }
```

Print total number of fields:

```
END          { nf = nf + NF }  
            { print nf }
```

Print total number of input characters:

```
END          { nc = nc + length($0) }  
            { print nc + NR }
```

(Adding **NR** includes in the total the number of newlines.)

Print the total number of lines that contain the string Asia:

```
/Asia/      { nlines++ }  
END        { print nlines }
```

(The statement “nlines++” has the same effect as “nlines = nlines + 1”.)

```
/Asia/      { nlines++ }  
END        { print nlines }
```

Error Messages

If you make an error in your **awk** program, you generally get an error message. For example, trying to run the program

```
$3 < 200 { print ( $1 ) }
```

generates the error messages

```
awk: syntax error at source line 1  
context is  
    $3 < 200 { print ( >>> $1 ) <<<  
awk: illegal statement at source line 1  
1 extra (
```

Some errors may be detected while your program is running. For example, if you try to divide a number by zero, **awk** stops processing and reports the input record number (NR) and the line number in the program.

Patterns

In a pattern-action statement, the pattern is an expression that selects the records for which the associated action is executed. This section describes the kinds of expressions that may be used as patterns.

BEGIN and END

BEGIN and END are two special patterns that give you a way to control initialization and wrap-up in an `awk` program. BEGIN matches before the first input record is read, so any statements in the action part of a BEGIN are done once, before the `awk` command starts to read its first input record. The pattern END matches the end of the input, after the last record has been processed.

The following `awk` program uses BEGIN to set the field separator to tab (`\t`) and to put column headings on the output. The field separator is stored in a built-in variable called FS. Although FS can be reset at any time, usually the only sensible place is in a BEGIN section, before any input has been read. The program's second `printf` statement, which is executed for each input line, formats the output into a table, neatly aligned under the column headings. The END action prints the totals. (Notice that a long line can be continued after a comma.)

```
BEGIN { FS = "\t"
        printf "%10s %6s %5s %s\n",
              "COUNTRY", "AREA", "POP", "CONTINENT" }
{ printf "%10s %6d %5d %s\n", $1, $2, $3, $4
  area = area + $2; pop = pop + $3 }
END   { printf "\n%10s %6d %5d\n", "TOTAL", area, pop }
```

With the file *countries* as input, this program produces

COUNTRY	AREA	POP	CONTINENT
USSR	8650	262	Asia
Canada	3852	24	North America
China	3692	866	Asia
USA	3615	219	North America
Brazil	3286	116	South America
Australia	2968	14	Australia
India	1269	637	Asia
Argentina	1072	26	South America
Sudan	968	19	Africa
Algeria	920	18	Africa
TOTAL	30292	2201	

Relational Expressions

An **awk** pattern can be any expression involving comparisons between strings of characters or numbers. To make comparisons, **awk** has six relational operators and two regular expression matching operators, `~` (tilde) and `!~`, which are discussed in the next section. Table 11.1 shows these operators and their meanings.

Table 11.1
awk Comparison Operators

Operator	Meaning
<code><</code>	less than
<code><=</code>	less than or equal to
<code>==</code>	equal to
<code>!=</code>	not equal to
<code>>=</code>	greater than or equal to
<code>></code>	greater than
<code>-</code>	matches
<code>!~</code>	does not match

In a comparison, if both operands are numeric, a numeric comparison is made; otherwise, the operands are compared as strings. (Every value might be either a number or a string; usually **awk** can tell what is intended. The section “Number or String?” contains more information about this.) Thus, the pattern

```
$3>100
```

selects lines where the third field exceeds 100, and the program

```
$1 >= "S"
```

selects lines that begin with the letters S through Z, namely,

```
USSR      8650  262  Asia
USA       3615  219  North America
Sudan     968   19   Africa
```

In the absence of any other information, **awk** treats fields as strings, so the program

```
$1 == $4
```

compares the first and fourth fields as strings of characters, and with the file **countries** as input, prints the single line for which this test succeeds:

```
Australia 2968 14 Australia
```

If both fields appear to be numbers, the comparisons are done numerically.

Regular Expressions

awk provides more powerful patterns for searching for strings of characters than the comparisons illustrated in the previous section. These patterns are called regular expressions and are like those in **grep(C)** in the *User's Reference*. The simplest regular expression is a string of characters enclosed in slashes, like

```
/Asia/
```

This program prints all input records that contain the substring Asia (If a record contains Asia as part of a larger string like Asian or Pan-Asiatic it is also printed.) In general, if *re* is a regular expression, then the pattern

```
/re/
```

matches any line that contains a substring specified by the regular expression *re*.

To restrict a match to a specific field, you use the matching operators `~` (matches) and `!~` (does not match). The program

```
$4 ~ /Asia/ { print $1 }
```

prints the first field of all lines in which the fourth field matches Asia while the program

```
$4 !~ /Asia/ { print $1 }
```

prints the first field of all lines in which the fourth field does not match Asia.

Patterns

In regular expressions, the symbols

```
\ ^ $ . [ ] * + ? ( ) |
```

are metacharacters with special meanings like the metacharacters in the UNIX system shell. For example, the metacharacters `^` and `$` match the beginning and end, respectively, of a string, and the metacharacter `.` (“dot”) matches any single character. Thus,

```
/^.$/
```

matches all records that contain exactly one character.

A group of characters enclosed in brackets matches any one of the enclosed characters; for example, `/[ABC]/` matches records containing any one of **A**, **B**, or **C** anywhere. Ranges of letters or digits can be abbreviated within brackets: `/[a-zA-Z]/` matches any single letter.

If the first character after the left bracket (`[`) is a caret (`^`), this complements the class so it matches any character not in the set: `/[^a-zA-Z]/` matches any non-letter. The program

```
$2 !~ /^ [0-9]+$/
```

prints all records in which the second field is not a string of one or more digits (`^` for beginning of string, `[0-9]+` for one or more digits, and `$` for end of string). Programs of this nature are often used for data validation.

Parentheses (`()`) are used for grouping and the symbol `|` is used for alternatives. The program

```
/(apple|cherry) (pie|tart)/
```

matches lines containing any one of the four substrings “apple pie,” “apple tart,” “cherry pie,” or “cherry tart.”

To turn off the special meaning of a metacharacter, precede it by a `\` (backslash). Thus, the program

```
/b\$/
```

prints all lines containing **b** followed by a dollar sign.

In addition to recognizing metacharacters, the **awk** command recognizes the following C programming language escape sequences within regular expressions and strings:

<code>\b</code>	backspace
<code>\f</code>	formfeed
<code>\n</code>	newline
<code>\r</code>	carriage return
<code>\t</code>	tab
<code>\ddd</code>	octal value <i>ddd</i>
<code>\"</code>	quotation mark
<code>\c</code>	any other character <i>c</i> literally

For example, to print all lines containing a tab, use the program

```
/\t/
```

awk interprets any string or variable on the right side of a `~` or `!~` as a regular expression. For example, we could have written the program

```
$2 !~ /^[0-9]+$/
```

as

```
BEGIN { digits = "[0-9]+" }
$2 !~ digits
```

Suppose you wanted to search for a string of characters like `^[0-9]+$`. When a literal quoted string like `"^[0-9]+$"` is used as a regular expression, one extra level of backslashes is needed to protect regular expression metacharacters. This is because one level of backslashes is removed when a string is originally parsed. If a backslash is needed in front of a character to turn off its special meaning in a regular expression, then that backslash needs a preceding backslash to protect it in a string.

For example, suppose we want to match strings containing **b** followed by a dollar sign. The regular expression for this pattern is `b\$`. If we want to create a string to represent this regular expression, we must add one more backslash: `"b\\$"`. The two regular expressions on each of the following lines are equivalent:

<code>x ~ "b\\\$"</code>	<code>x ~ /b\\$/</code>
<code>x ~ "b\\$"</code>	<code>x ~ /b\$/</code>
<code>x ~ "b\\$"</code>	<code>x ~ /b\\$/</code>
<code>x ~ "\\t"</code>	<code>x ~ /\t/</code>

Patterns

The precise form of regular expressions and the substrings they match is given in Table 11.2. The unary operators `*`, `+`, and `?` have the highest precedence, with concatenation next, and then alternation `|`. All operators are left associative. The `r` stands for any regular expression.

Table 11.2
awk Regular Expressions

Expression	Matches
<code>c</code>	any non-metacharacter <code>c</code>
<code>\c</code>	character <code>c</code> literally
<code>^</code>	beginning of string
<code>\$</code>	end of string
<code>.</code>	any character but newline
<code>[s]</code>	any character in set <code>s</code>
<code>[^s]</code>	any character not in set <code>s</code>
<code>r*</code>	zero or more <code>r</code> 's
<code>r+</code>	one or more <code>r</code> 's
<code>r?</code>	zero or one <code>r</code>
<code>(r)</code>	<code>r</code>
<code>r₁r₂</code>	<code>r₁</code> then <code>r₂</code> (concatenation)
<code>r₁ r₂</code>	<code>r₁</code> or <code>r₂</code> (alternation)

Combinations of Patterns

A compound pattern combines simpler patterns with parentheses and the logical operators `||` (or), `&&` (and), and `!` (not). For example, suppose we want to print all countries in Asia with a population of more than 500 million. The following program does this by selecting all lines in which the fourth field is `Asia` and the third field exceeds 500:

```
$4 == "Asia" && $3 > 500
```

The program

```
$4 == "Asia" || $4 == "Africa"
```

selects lines with `Asia` or `Africa` as the fourth field. Another way to write the latter query is to use a regular expression with the alternation operator `|`:

```
$4 ~ /^(Asia|Africa)$/
```

The negation operator `!` has the highest precedence, then `&&`, and finally `||`. The operators `&&` and `||` evaluate their operands from left to right; evaluation stops as soon as truth or falsehood is determined.

Pattern Ranges

A pattern range consists of two patterns separated by a comma, as in

```
pat1, pat2    { . . . }
```

In this case, the action is performed for each line between an occurrence of *pat*₁ and the next occurrence of *pat*₂ (inclusive). As an example, the pattern

```
/Canada/, /Brazil/
```

matches lines starting with the first line that contains the string `Canada`, up through the next occurrence of the string `Brazil`:

Canada	3852	24	North America
China	3692	866	Asia
USA	3615	219	North America
Brazil	3286	116	South America

Similarly, because `FNR` is the number of the current record in the current input file (and `FILENAME` is the name of the current input file), the program

```
FNR == 1, FNR == 5 { print FILENAME, $0 }
```

prints the first five records of each input file with the name of the current input file prepended.

Actions

In a pattern-action statement, the action determines what is to be done with the input records that the pattern selects. Actions frequently are simple printing or assignment statements, but they may also be a combination of one or more statements. This section describes the statements that can make up actions.

Built-in Variables

Table 11.3 lists the built-in variables that **awk** maintains. Some of these we have already met; others are used in this and later sections.

Table 11.3
awk Built-in Variables

Variable	Meaning	Default
ARGC	number of command-line arguments	-
ARGV	array of command-line arguments	-
FILENAME	name of current input file	-
FNR	record number in current file	-
FS	input field separator	blank&tab
NF	number of fields in current record	-
NR	number of records read so far	-
OFMT	output format for numbers	%.6g
OFS	output field separator	blank
ORS	output record separator	newline
RS	input record separator	newline
RSTART	index of first character matched by <code>match()</code>	-
RLENGTH	length of string matched by <code>match()</code>	-
SUBSEP	subscript separator	"\034"

Arithmetic

Actions can use conventional arithmetic expressions to compute numeric values. As a simple example, suppose we want to print the population density for each country in the file *countries*. Because the second field is

the area in thousands of square miles, and the third field is the population in millions, the expression `1000 * $3 / $2` gives the population density in people per square mile. The program

```
{ printf "%10s %6.1f\n", $1, 1000 * $3 / $2 }
```

applied to the file *countries* prints the name of each country and its population density. The output looks like this:

```
USSR    30.3
Canada  6.2
China   234.6
USA     60.6
Brazil  35.3
Australia 4.7
India   502.0
Argentina 24.3
Sudan   19.6
Algeria 19.6
```

Arithmetic is done internally in floating point. The arithmetic operators are `+`, `-`, `*`, `/`, `%` (remainder), and `^` (exponentiation; `**` is a synonym). Arithmetic expressions can be created by applying these operators to constants, variables, field names, array elements, functions, and other expressions, all of which are discussed later. Note that `awk` recognizes and produces scientific (exponential) notation: `1e6`, `1E6`, `10e5`, and `1000000` are numerically equal.

`awk` has assignment statements like those found in the C programming language. The simplest form is the assignment statement

$$v = e$$

where v is a variable or field name, and e is an expression. For example, to compute the number of Asian countries and their total populations, we could write

```
$4 == "Asia" { pop = pop + $3; n = n + 1 }
END          { print "population of", n,
                "Asian countries in millions is", pop }
```

Applied to *countries*, this program produces

```
population of 3 Asian countries in millions is 1765
```

The action associated with the pattern `$4 == "Asia"` contains two assignment statements, one to accumulate population and the other to count

Actions

countries. The variables are not explicitly initialized, yet everything works properly because `awk` initializes each variable with the string value "" and the numeric value 0.

The assignments in the previous program can be written more concisely using the operators `+=` and `++` as follows:

```
$4 == "Asia" { pop += $3; ++n }
```

The operator `+=` is borrowed from the C programming language:

```
pop += $3
```

It has the same effect as

```
pop = pop + $3
```

but the `+=` operator is shorter and runs faster. The same is true of the `++` operator, which adds one to a variable.

The abbreviated assignment operators are `+=`, `-=`, `*=`, `/=`, `%=`, and `^=`. Their meanings are similar. For example,

```
v op= e
```

has the same effect as

```
v = v op e.
```

The increment operators are `++` and `-`. As in C, they may be used as prefix (`++x`) or postfix (`x++`) operators. If `x` is 1, then `i=++x` increments `x`, then sets `i` to 2, while `i=x++` sets `i` to 1, then increments `x`. An analogous interpretation applies to prefix and postfix `-`.

Assignment and increment and decrement operators may all be used in arithmetic expressions.

We use default initialization to advantage in the following program, which finds the country with the largest population:

```
maxpop < $3 { maxpop = $3; country = $1 }  
END { print country, maxpop }
```

Note, however, that this program is not correct if all values of \$3 are negative.

awk provides the built-in arithmetic functions shown in Table 11.4.

Table 11.4
awk Built-in Arithmetic Functions

Function	Value Returned
atan2 (y,x)	arctangent of y/x in the range $-\pi$ to π
cos (x)	cosine of x , with x in radians
exp (x)	exponential function of x
int (x)	integer part of x truncated towards 0
log (x)	natural logarithm of x
rand ()	random number between 0 and 1
sin (x)	sine of x , with x in radians
sqrt (x)	square root of x
srand (x)	x is new seed for rand ()

Both x and y are arbitrary expressions. The function **rand**() returns a pseudo-random floating point number in the range (0,1), and **srand**(x) can be used to set the seed of the generator. If **srand**() has no argument, the seed is derived from the time of day.

Strings and String Functions

A string constant is created by enclosing a sequence of characters inside quotation marks, as in "abc" or "hello, everyone". String constants may contain the C programming language escape sequences for special characters listed in "Regular Expressions" in this chapter.

String expressions are created by concatenating constants, variables, field names, array elements, functions, and other expressions. The program

```
{ print NR ":" $0 }
```

prints each record preceded by its record number and a colon, with no blanks. The three strings representing the record number, the colon, and the record are concatenated, and the resulting string is printed. The concatenation operator has no explicit representation other than juxtaposition.

awk provides the built-in string functions shown in Table 11.5. In this table, r represents a regular expression (either as a string or as $/r/$), s and t are string expressions, and n and p are integers.

Actions

Table 11.5
awk Built-in String Functions

Function	Description
gsub (<i>r, s</i>)	substitutes <i>s</i> for <i>r</i> globally in current record, returns number of substitutions
gsub (<i>r, s, t</i>)	substitutes <i>s</i> for <i>r</i> globally in string <i>t</i> , returns number of substitutions
index (<i>s, t</i>)	returns position of string <i>t</i> in <i>s</i> , 0 if not present
length (<i>s</i>)	returns length of <i>s</i>
match (<i>s, r</i>)	returns the position in <i>s</i> where <i>r</i> occurs, 0 if not present
split (<i>s, a</i>)	splits <i>s</i> into array <i>a</i> on FS, returns number of fields
split (<i>s, a, r</i>)	splits <i>s</i> into array <i>a</i> on <i>r</i> , returns number of fields
sprintf (<i>fmt, expr-list</i>)	returns <i>expr-list</i> formatted according to format string <i>fmt</i>
sub (<i>r, s</i>)	substitutes <i>s</i> for first <i>r</i> in current record, returns number of substitutions
sub (<i>r, s, t</i>)	substitutes <i>s</i> for first <i>r</i> in <i>t</i> , returns number of substitutions
substr (<i>s, p</i>)	returns suffix of <i>s</i> starting at position <i>p</i>
substr (<i>s, p, n</i>)	returns substring of <i>s</i> of length <i>n</i> starting at position <i>p</i>

The functions **sub** and **gsub** are patterned after the substitute command in the text editor **ed**(c), which can be found in both the *User's Reference Manual*. The function **gsub**(*r, s, t*) replaces successive occurrences of substrings matched by the regular expression *r* with the replacement string *s* in the target string *t*. (As in **ed**, the left-most match is used and is made as long as possible.) **gsub** returns the number of substitutions made. The function **gsub**(*r, s*) is a synonym for **gsub**(*r, s, \$0*). For example, the program

```
{ gsub(/USA/, "United States"); print }
```

transcribes its input, replacing occurrences of USA by United States. The **sub** functions are similar, except that they only replace the first matching substring in the target string.

The function `index(s,t)` returns the left-most position where the string `t` begins in `s`, or zero if `t` does not occur in `s`. The first character in a string is at position 1. For example,

```
index("banana", "an")
```

returns 2.

The `length` function returns the number of characters in its argument string; thus,

```
{ print length($0), $0 }
```

prints each record, preceded by its length. (`$0` does not include the input record separator.) The program

```
length($1) > max { max = length($1); name = $1 }
END             { print name }
```

applied to the file `countries` prints the longest country name: **Australia**.

The `match(s,r)` function returns the position in string `s` where regular expression `r` occurs, or 0 if it does not occur. This function also sets two built-in variables `RSTART` and `RLENGTH`. `RSTART` is set to the starting position of the match in the string; this is the same value as the returned value. `RLENGTH` is set to the length of the matched string. (If a match does not occur, `RSTART` is 0, and `RLENGTH` is -1.) For example, the following program finds the first occurrence of the letter `i`, followed by at most one character, followed by the letter `a` in a record:

```
{ if (match($0, /i.?a/))
    print RSTART, RLENGTH, $0 }
```

It produces the following output on the file `countries`:

217	2	USSR	8650	262	Asia
26	3	Canada	3852	24	North America
3	3	China	3692	866	Asia
24	3	USA	3615	219	North America
27	3	Brazil	3286	116	South America
8	2	Australia	2968	14	Australia
4	2	India	1269	637	Asia
7	3	Argentina	1072	26	South America
17	3	Sudan	968	19	Africa
6	2	Algeria	920	18	Africa

Actions

Note

match() matches the left-most longest matching string. For example, with the record

```
AsiaaaaAsiaaaaaa
```

as input, the program

```
{ if (match($0, /a+/)) print RSTART, RLENGTH, $0 }
```

matches the first string of **a**'s and sets **RSTART** to 4 and **RLENGTH** to 3.

Notice the following function:

```
sprintf(format, expr 1, expr 2, ...,
```

This returns (without printing) a string containing:

```
expr 1, expr 2, ...,
```

expr_n formatted according to the **printf** specifications in the string *format*. “The printf Statement” in this chapter contains a complete specification of the format conventions. The statement

```
x = sprintf("%10s %6d", $1, $2)
```

assigns to **x** the string produced by formatting the values of **\$1** and **\$2** as a 10-character string and a decimal number in a field of width at least six; **x** may be used in any subsequent computation.

The function **substr**(*s*, *p*, *n*) returns the substring of *s* that begins at position *p* and is at most *n* characters long. If **substr**(*s*, *p*) is used, the substring goes to the end of *s*; that is, it consists of the suffix of *s* beginning at position *p*. For example, we could abbreviate the country names in *countries* to their first three characters by invoking the program

```
{ $1 = substr($1, 1, 3); print }
```

This produces

```

USS 8650 262 Asia
Can 3852 24 North America
Chi 3692 866 Asia
USA 3615 219 North America
Bra 3286 116 South America
Aus 2968 14 Australia
Ind 1269 637 Asia
Arg 1072 26 South America
Sud 968 19 Africa
Alg 920 18 Africa

```

Note that setting **\$1** in the program forces **awk** to recompute **\$0** and, therefore, the fields are separated by blanks (the default value of **OFS**), not by tabs.

Strings are stuck together (concatenated) merely by writing them one after another in an expression. For example, when invoked on file *countries*,

```

      { s = s substr($1, 1, 3) " " }
END { print s }

```

prints

```

USS Can Chi USA Bra Aus Ind Arg Sud Alg

```

by building **s** up, a piece at a time, from an initially empty string.

Field Variables

The fields of the current record can be referred to by the field variables **\$1**, **\$2**, . . . , **\$NF**. Field variables share all of the properties of other variables: they can be used in arithmetic or string operations, and they can have values assigned to them. So, for example, you can divide the second field of the file *countries* by 1000 to convert the area from thousands to millions of square miles

```

{ $2 /= 1000; print }

```

or assign a new string to a field:

```

BEGIN          { FS = OFS = "\t" }
$4 == "North America" { $4 = "NA" }
$4 == "South America" { $4 = "SA" }
                { print }

```

Actions

The `BEGIN` action in this program resets the input field separator `FS` and the output field separator `OFS` to a tab. Notice that the *print* in the fourth line of the program prints the value of `$0` after it has been modified by previous assignments.

Fields can be accessed by expressions. For example, `$(NF-1)` is the second to last field of the current record. The parentheses are needed to show that the value of `$(NF-1)` is 1 less than the value in the last field.

A field variable referring to a nonexistent field, for example, `$(NF+1)`, has as its initial value the empty string. A new field can be created, however, by assigning a value to it. For example, the following program invoked on the file *countries* creates a fifth field giving the population density:

```
BEGIN { FS = OFS = "\t" }
      { $5 = 1000 * $3 / $2; print }
```

The number of fields can vary from record to record, but there is usually an implementation limit of 100 fields per record.

Number or String?

Variables, fields, and expressions can have both a numeric value and a string value. They take on numeric or string values according to context. For example, in the context of an arithmetic expression like

```
pop += $3
```

`pop` and `$3` must be treated numerically, so their values can be coerced to numeric type if necessary.

In a string context like

```
print $1 ":" $2
```

`$1` and `$2` must be strings to be concatenated, so they can be coerced if necessary.

In an assignment $v = e$ or $v\ op = e$, the type of v becomes the type of e . In an ambiguous context like

```
$1 == $2
```

the type of the comparison depends on whether the fields are numeric or string, and this can only be determined when the program runs; it may well differ from record to record.

In comparisons, if both operands are numeric, the comparison is numeric; otherwise, operands are coerced to strings, and the comparison is made on the string values. All field variables are of type string; in addition, each field that contains only a number is also considered numeric. This determination is done at run time. For example, the comparison "\$1 == \$2" succeeds on any pair of the inputs

```
1      1.0    +1      0.1e+1    10E-1    001
```

but fails on the inputs

```
(null)      0
(null)      0.0
0a          0
1e50       1.0e50
```

There are two idioms for coercing an expression of one type to the other:

```
number ""    concatenate a null string to a number to coerce it
                to type string
string + 0    add zero to a string to coerce it to type numeric
```

Thus, to force a string comparison between two fields, say

```
$1 "" == $2 ""
```

The numeric value of a string is the value of any prefix of the string that looks numeric; thus the value of **12.34x** is 12.34, while the value of **x12.34** is zero. The string value of an arithmetic expression is computed by formatting the string with the output format conversion OFMT.

Uninitialized variables have numeric value 0 and string value "". Nonexistent fields and fields that are explicitly null have only the string value ""; they are not numeric.

Control Flow Statements

awk provides **if-else**, **while**, **do-while**, and **for** statements, and statement grouping with braces, as in the C programming language.

Actions

The **if** statement syntax is

```
if (expression) statement 1 else statement 2
```

The *expression* acting as the conditional has no restrictions; it can include the relational operators <, <=, >, >=, ==, and !=; the regular expression matching operators ~ and !~; the logical operators ||, &&, and !; juxtaposition for concatenation; and parentheses for grouping.

In the **if** statement, the *expression* is first evaluated. If it is non-zero and non-null, *statement* ₁ is executed; otherwise *statement* ₂ is executed. The **else** part is optional.

A single statement can always be replaced by a statement list enclosed in braces. The statements in the statement list are terminated by newlines or semicolons.

Rewriting the maximum population program from “Arithmetic Functions” with an **if** statement results in

```
{    if (maxpop < $3) {
        maxpop = $3
        country = $1
    }
}
END { print country, maxpop }
```

The **while** statement is exactly that of the C programming language:

```
while (expression) statement
```

The *expression* is evaluated; if it is non-zero and non-null, the *statement* is executed, and the *expression* is tested again. The cycle repeats as long as the *expression* is non-zero. For example, to print all input fields one per line,

```
{    i = 1
    while (i <= NF) {
        print $i
        i++
    }
}
```

The **for** statement is like that of the C programming language:

```
for (expression 1; expression; expression 2) statement
```

It has the same effect as

```

expression 1
while (expression) {
    statement
    expression 2
}

```

so

```
{ for (i = 1; i <= NF; i++) print $i }
```

does the same job as the **while** example above. An alternate version of the **for** statement is described in the next section.

The **do** statement has the form

```
do statement while (expression)
```

The *statement* is executed repeatedly until the value of the *expression* becomes zero. Because the test takes place after the execution of the *statement* (at the bottom of the loop), it is always executed at least once. As a result, the **do** statement is used much less often than **while** or **for**, which test for completion at the top of the loop.

The following example of a **do** statement prints all lines except those between *start* and *stop*.

```

/start/ {
    do {
        getline x
    } while (x !~ /stop/)
}
{ print }

```

The **break** statement causes an immediate exit from an enclosing **while** or **for**; the **continue** statement causes the next iteration to begin. The **next** statement causes **awk** to skip immediately to the next record and begin matching patterns starting from the first pattern-action statement.

The **exit** statement causes the program to behave as if the end of the input had occurred; no more input is read, and the END action, if any, is executed. Within the END action,

```
exit expr
```

causes the program to return the value of *expr* as its exit status. If there is no *expr*, the exit status is zero.

Actions

Arrays

awk provides one-dimensional arrays. Arrays and array elements need not be declared; like variables, they spring into existence by being mentioned. An array subscript may be a number or a string.

As an example of a conventional numeric subscript, the statement

```
x[NR] = $0
```

assigns the current input line to the NRth element of the array *x*. In fact, it is possible in principle (though perhaps slow) to read the entire input into an array with the **awk** program

```
{ x[NR] = $0 }
END { ...processing ... }
```

The first action merely records each input line in the array *x*, indexed by line number; processing is done in the END statement.

Array elements may also be named by nonnumeric values. For example, the following program accumulates the total population of Asia and Africa into the associative array *pop*. The END action prints the total population of these two continents.

```
/Asia/   { pop["Asia"] += $3 }
/Africa/ { pop["Africa"] += $3 }
END      { print "Asian population in millions is", pop["Asia"]
           print "African population in millions is",
           pop["Africa"] }
```

On the file *countries*, this program generates

```
Asian population in millions is 1765
African population in millions is 37
```

In this program, if we had used *pop[Asia]* instead of *pop["Asia"]*, the expression would have used the value of the variable *Asia* as the subscript, and because the variable is uninitialized, the values would have been accumulated in *pop[""]*.

Suppose our task is to determine the total area in each continent of the file *countries*. Any expression can be used as a subscript in an array reference. Thus,

```
area[$4] += $2
```

uses the string in the fourth field of the current input record to index the array `area` and in that entry accumulates the value of the second field:

```
BEGIN    { FS = "\t" }
          { area[$4] += $2 }
END      { for (name in area)
          print name, area[name] }
```

Invoked on the file *countries*, this program produces

```
Africa 1888
North America 7467
South America 4358
Asia 13611
Australia 2968
```

This program uses a form of the **for** statement that iterates over all defined subscripts of an array:

for (*i* in *array*) *statement*

executes *_statement* with the variable *i* set in turn to each value of *i* for which *array*[*i*] has been defined. The loop is executed once for each defined subscript, which are chosen in a random order. Results are unpredictable when *i* or *array* is altered during the loop.

awk does not provide multi-dimensional arrays, but it does permit a list of subscripts. They are combined into a single subscript with the values separated by an unlikely string (stored in the variable `SUBSEP`). For example,

```
for (i = 1; i <= 10; i++)
  for (j = 1; j <= 10; j++)
    arr[i,j] = ...
```

creates an array that behaves like a two-dimensional array; the subscript is the concatenation of *i*, `SUBSEP`, and *j*.

You can determine whether a particular subscript *i* occurs in an array *arr* by testing the condition *i* in *arr*, as in

```
if ("Africa" in area) ...
```

This condition performs the test without the side effect of creating `area["Africa"]`, which would happen if we used

```
if (area["Africa"] != "") ...
```

Actions

Note that neither is a test of whether the array **area** contains an element with value "Africa".

It is also possible to split any string into fields in the elements of an array using the built-in function **split**. The function

```
split("s1:s2:s3", a, ":")
```

splits the string `s1:s2:s3` into three fields, using the separator `:`, and stores `s1` in `a[1]`, `s2` in `a[2]`, and `s3` in `a[3]`. The number of fields found, here three, is returned as the value of **split**. The third argument of **split** is a regular expression to be used as the field separator. If the third argument is missing, FS is used as the field separator.

An array element may be deleted with the **delete** statement:

```
delete arrayname[subscript]
```

User-Defined Functions

awk provides user-defined functions. A function is defined as

```
function name(argument-list) {  
    statements  
}
```

The definition can occur anywhere a pattern-action statement can. The argument list is a list of variable names separated by commas; within the body of the function, these variables refer to the actual parameters when the function is called. There must be no space between the function name and the left parenthesis of the argument list when the function is called; otherwise it looks like a concatenation. For example, the following program defines and tests the usual recursive factorial function (of course, using some input other than the file *countries*):

```
function fact(n) {  
    if (n <= 1)  
        return 1  
    else  
        return n * fact(n-1)  
}  
{ print $1 "!" is " fact($1) }
```

Array arguments are passed by reference, as in C, so it is possible for the function to alter array elements or create new ones. Scalar arguments are

passed by value, however, so the function cannot affect their values outside. Within a function, formal parameters are local variables, but all other variables are global. (You can have any number of extra formal parameters that are used purely as local variables.) The `return` statement is optional, but the returned value is undefined if it is not included.

Some Lexical Conventions

Comments may be placed in `awk` programs: they begin with the character `#` and end at the end of the line, as in

```
print x, y      # this is a comment
```

Statements in an `awk` program normally occupy a single line. Several statements may occur on a single line if they are separated by semicolons. A long statement may be continued over several lines by terminating each continued line by a backslash. (It is not possible to continue a `"..."` string.) This explicit continuation is rarely necessary, however, since statements continue automatically after the operators `&&` and `||` or if the line ends with a comma (for example, as might occur in a `print` or `printf` statement).

Several pattern-action statements may appear on a single line if separated by semicolons.

Output

The **print** and **printf** statements are the two primary constructs that generate output. The **print** statement is used to generate simple output; **printf** is used for more carefully formatted output. Like the shell, **awk** lets you redirect output so that output from **print** and **printf** can be directed to files and pipes. This section describes the use of these two statements.

The print Statement

The statement

```
print expr 1, expr 2, ..., expr n
```

prints the string value of each expression separated by the output field separator followed by the output record separator. The statement

```
print
```

is an abbreviation for

```
print $0
```

To print an empty line use

```
print ""
```

Output Separators

The output field separator and record separator are held in the built-in variables **OFS** and **ORS**. Initially, **OFS** is set to a single blank and **ORS** to a single newline, but these values can be changed at any time. For example, the following program prints the first and second fields of each record with a colon between the fields and two newlines after the second field:

```
BEGIN { OFS = ":"; ORS = "\n\n" }
       { print $1, $2 }
```

Notice that

```
{ print $1 $2 }
```

prints the first and second fields with no intervening output field separator, because \$1 \$2 is a string consisting of the concatenation of the first two fields.

11

The printf Statement

awk's printf statement is the same as that in **C** except that the ***** format specifier is not supported. The **printf** statement has the general form

```
printf format, expr1, expr2, ..., exprn
```

where *format* is a string that contains both information to be printed and specifications on what conversions are to be performed on the expressions in the argument list, as in Table 11.6. Each specification begins with a %, ends with a letter that determines the conversion, and may include

- left-justify expression in its field
- width* pad field to this width as needed; fields that begin with a leading 0 are padded with zeros
- .prec* maximum string width or digits to right of decimal point

Output

Table 11.6
awk printf Conversion Characters

Character	Prints Expression as
c	single character
d	decimal number
e	[-.]d.dddE[+-]dd
f	[-.]ddd.ddd
g	e or f conversion, whichever is shorter, with nonsignificant zeros suppressed
o	unsigned octal number
s	string
x	unsigned hexadecimal number
%	print a %; no argument is converted

Here are some examples of `printf` statements with the corresponding output:

```
printf "%d", 99/2           49
printf "%e", 99/2         4.950000e+01
printf "%f", 99/2         49.500000
printf "%6.2f", 99/2      49.50
printf "%g", 99/2         49.5
printf "%o", 99           143
printf "%06o", 99         000143
printf "%x", 99           63
printf "%s", "January"   |January|
printf "%10s", "January"  |  January|
printf "%-10s", "January" |January  |
printf "%.3s", "January" |Jan|
printf "%10.3s", "January" |          Jan|
printf "%-10.3s", "January" |Jan          |
printf "%%"              %
```

The default output format of numbers is `%.6g`; this can be changed by assigning a new value to `OFMT`. `OFMT` also controls the conversion of numeric values to strings for concatenation and creation of array subscripts.

Output into Files

It is possible to print output into files, instead of to the standard output, by using the `>` and `>>` redirection operators. For example, the following program invoked on the file *countries* prints all lines where the population (third field) is bigger than 100 into a file called *bigpop*, and all other lines into *smallpop*:

```
$3 > 100 { print $1, $3 >"bigpop" }
$3 <= 100 { print $1, $3 >"smallpop" }
```

Notice that the filenames have to be quoted; without quotes, *bigpop* and *smallpop* are merely uninitialized variables. If the output filenames were created by an expression, they would also have to be enclosed in parentheses:

```
$4 ~ /North America/ { print $1 > ("tmp" FILENAME) }
```

This is because the `>` operator has higher precedence than concatenation; without parentheses, the concatenation of *tmp* and *FILENAME* would not work.

Note

Files are opened once in an `awk` program. If `>` is used to open a file, its original contents are overwritten. But if `>>` is used to open a file, its contents are preserved and the output is appended to the file. Once the file has been opened, the two operators have the same effect.

Output into Pipes

It is also possible to direct printing into a pipe with a command on the other end, instead of into a file. The statement

```
print | "command-line"
```

causes the output of `print` to be piped into the *command-line*.

Output

Although we have shown them here as literal strings enclosed in quotes, the *command-line* and file names can come from variables, and the return values from functions, for instance.

Suppose we want to create a list of continent-population pairs, sorted alphabetically by continent. The `awk` program below accumulates the population values in the third field for each of the distinct continent names in the fourth field in an array called *pop*. Then it prints each continent and its population, and pipes this output into the `sort` command.

```
BEGIN { FS = "\t" }
        { pop[$4] += $3 }
END    { for (c in pop)
        print c ":" pop[c] | "sort" }
```

Invoked on the file *countries*, this program yields

```
Africa:37
Asia:1765
Australia:14
North America:243
South America:142
```

In all of these `print` statements involving redirection of output, the files or pipes are identified by their names (that is, the pipe above is literally named *sort*), but they are created and opened only once in the entire run. So, in the last example, for all *c* in *pop*, only one `sort` pipe is open.

There is a limit to the number of files that can be open simultaneously. The statement `close(file)` closes a file or pipe; *file* is the string used to create it in the first place, as in

```
close("sort")
```

When opening or closing a file, different strings are different commands.

Input

The most common way to give input to an `awk` program is to name on the command line the file(s) that contains the input. This is the method we have been using in this chapter. However, there are several other methods we could use, each of which this section describes.

Files and Pipes

You can provide input to an `awk` program by putting the input data into a file, say `awkdata`, and then executing

```
awk 'program' awkdata
```

`awk` reads its standard input if no filenames are given (see “Usage” in this chapter); thus, a second common arrangement is to have another program pipe its output into `awk`. For example, `grep(C)`, in the *User's Reference*, selects input lines containing a specified regular expression, but it can do so faster than `awk`, because this is the only thing it does. We could, therefore, invoke the pipe

```
grep 'Asia' countries | awk '...'
```

`grep` quickly finds the lines containing Asia and passes them on to the `awk` program for subsequent processing.

Input Separators

With the default setting of the field separator `FS`, input fields are separated by blanks or tabs, and leading blanks are discarded, so each of these lines has the same first field:

```
    field1  field2
field1
field1
```

When the field separator is a tab, however, leading blanks are not discarded.

Input

The field separator can be set to any regular expression by assigning a value to the built-in variable FS. For example,

```
BEGIN { FS = "(, [ \\t]*)|([ \\t]+)" }
```

sets it to an optional comma followed by any number of blanks and tabs. FS can also be set on the command line with the `-F` argument:

```
awk -F '(, [ \\t]*)|([ \\t]+)' '...'
```

behaves the same as the previous example. Regular expressions used as field separators match the left-most longest occurrences (as in `sub()`), but they do not match null strings.

Multi-Line Records

Records are normally separated by newlines, so that each line is a record, but this too can be changed, though only in a limited way. If the built-in record separator variable RS is set to the empty string, as in

```
BEGIN { RS = "" }
```

then input records can be several lines long; a sequence of empty lines separates records. A common way to process multiple-line records is to use

```
BEGIN { RS = ""; FS = "\n" }
```

to set the record separator to an empty line and the field separator to a newline. There is a limit, however, on how long a record can be; it is usually about 2500 characters. “The `getline` Function” and “Cooperation with the Shell” in this chapter show other examples of processing multi-line records.

The `getline` Function

`awk`'s facility for automatically breaking its input into records that are more than one line long is not adequate for some tasks. For example, if records are not separated by blank lines, but by something more complicated, merely setting RS to null does not work. In such cases, it is necessary to manage the splitting of each record into fields in the program. Here are some suggestions.

The function `getline` can be used to read input either from the current input or from a file or pipe, by using redirection in a manner analogous to

printf. By itself, **getline** fetches the next input record and performs the normal field-splitting operations on it. It sets **NF**, **NR**, and **FNR**. **getline** returns 1 if there was a record present, 0 if the end-of-file was encountered, and -1 if some error occurred (such as failure to open a file).

To illustrate, suppose we have input data consisting of multi-line records, each of which begins with a line beginning with **START** and ends with a line beginning with **STOP**. The following **awk** program processes these multi-line records, a line at a time, putting the lines of the record into consecutive entries of an array

```
f[1] f[2] ... f[nf]
```

Once the line containing **STOP** is encountered, the record can be processed from the data in the **f** array:

```
/^START/ {
    f[nf=1] = $0
    while (getline && $0 !~ /^STOP/)
        f[++nf] = $0
    # now process the data in f[1]...f[nf]
    ...
}
```

Notice that this code uses the fact that **&&** evaluates its operands left to right and stops as soon as one is true. The same job can also be done by the following program:

```
/^START/ && nf==0 { f[nf=1] = $0 }
nf > 1           { f[++nf] = $0 }
/^STOP/         { # now process the data in f[1]...f[nf]
                  ...
                  nf = 0
}
```

The statement

```
getline x
```

reads the next record into the variable *x*. No splitting is done; **NF** is not set. The statement

```
getline <"file"
```

reads from *file* instead of the current input. It has no effect on **NR** or **FNR**, but field splitting is performed, and **NF** is set. The statement

```
getline x <"file"
```

Input

gets the next record from **file** into **x**; no splitting is done, and NF, NR and FNR are untouched.

Note

If a filename is an expression, it should be in parentheses for evaluation:

```
while ( getline x < (ARGV[1] ARGV[2]) ) { ... }
```

This is because the **<** has precedence over concatenation. Without parentheses, a statement such as

```
getline x < "tmp" FILENAME
```

sets **x** to read the file *tmp* <value of FILENAME>. Also, if you use this **getline** statement form, a statement like

```
while ( getline x < file ) { ... }
```

loops forever if the file cannot be read, because **getline** returns -1, not zero, if an error occurs. A better way to write this test is

```
while ( getline x < file > 0 ) { ... }
```

It is also possible to pipe the output of another command directly into **getline**. For example, the statement

```
while ("who" | getline)
    n++
```

executes *who* and pipes its output into *getline*. Each iteration of the *while* loop reads one more line and increments the variable *n*, so after the *while* loop terminates, *n* contains a count of the number of users. Similarly, the statement

```
"date" | getline d
```

pipes the output of *date* into the variable *d*, thus setting *d* to the current date. Table 11.7 summarizes the **getline** function.

Table 11.7
getline Function

Form	Sets
<code>getline</code>	<code>\$0, NF, NR, FNR</code>
<code>getline var</code>	<code>var, NR, FNR</code>
<code>getline <file</code>	<code>\$0, NF</code>
<code>getline var <file</code>	<code>var</code>
<code>cmd getline</code>	<code>\$0, NF</code>
<code>cmd getline var</code>	<code>var</code>

Command-Line Arguments

The command-line arguments are available to an `awk` program: the array `ARGV` contains the elements `ARGV[0], . . . , ARGV[ARGC-1]`; as in C, `ARGC` is the count. `ARGV[0]` is the name of the program (generally `awk`); the remaining arguments are whatever was provided (excluding the program and any optional arguments).

Input

The following command line contains an **awk** program that echoes the arguments that appear after the program name:

```
awk '
BEGIN {
    for (i = 1; i < ARGV; i++)
        printf "%s ", ARGV[i]
    printf "\n"
}' $*
```

The arguments may be modified or added to; **ARGC** may be altered. As each input file ends, **awk** treats the next non-null element of **ARGV** (up to the current value of **ARGC-1**) as the name of the next input file.

There is one exception to the rule that an argument is a filename: if it is of the form

var=value

then the variable *var* is set to the value *value* as if by assignment. Such an argument is not treated as a filename. If *value* is a string, no quotes are needed.

Using awk with Other Commands and the Shell

awk gains its greatest power when it is used in conjunction with other programs. Here we describe some of the ways in which **awk** programs cooperate with other commands.

The system Function

The built-in function **system** (*command-line*) executes the command *command-line*, which may well be a string computed by, for example, the built-in function **sprintf**. The value returned by **system** is the return status of the command executed.

For example, the program

```
$1 = "#include" { gsub(/[<>"/, "", $2); system("cat " $2) }
```

calls the command *cat* to print the file named in the second field of every input record whose first field is *#include*, after stripping any *<*, *>*, or *"* that might be present.

Cooperation with the Shell

In all the examples thus far, the **awk** program was in a file and fetched from there using the *-f* flag, or it appeared on the command line enclosed in single quotes, as in

```
awk '{ print $1 }' ...
```

Since **awk** uses many of the same characters as the shell does, such as *\$* and *"*, surrounding the **awk** program with single quotes ensures that the shell passes the entire program unchanged to the **awk** interpreter.

Using awk with Other Commands and the Shell

Now, consider writing a command **addr** that searches a file *addresslist* for name, address, and telephone information. Suppose that *addresslist* contains names and addresses in which a typical entry is a multi-line record such as

```
G. R. Emlin
600 Mountain Avenue
Murray Hill, NJ 07974
201-555-1234
```

Records are separated by a single blank line.

We want to search the address list by issuing commands like

```
addr Emlin
```

That is easily done by a program of the form

```
awk '
BEGIN { RS = "" }
/Emlin/
' addresslist
```

The problem is how to get a different search pattern into the program each time it is run.

There are several ways to do this. One way is to create a file called *addr* that contains

```
awk '
BEGIN { RS = "" }
/'$1'/
' addresslist
```

The quotes are critical here: the **awk** program is only one argument, even though there are two sets of quotes, because quotes do not nest. The **\$1** is outside the quotes, visible to the shell, which then replaces it by the pattern *Emlin* when the command **addr Emlin** is invoked. On a UNIX system, **addr** can be made executable by changing its mode with the following command: **chmod +x addr**.

A second way to implement **addr** relies on the fact that the shell substitutes for \$ parameters within double quotes:

```
awk "
BEGIN { RS = "\" }
/$1/
" addresslist
```

Here we must protect the quotes defining RS with backslashes so that the shell passes them on to **awk**, uninterpreted by the shell. \$1 is recognized as a parameter, however, so the shell replaces it by the pattern when the command **addr pattern** is invoked.

A third way to implement **addr** is to use ARGV to pass the regular expression to an **awk** program that explicitly reads through the address list with **getline**:

```
awk '
BEGIN { RS = ""
       while (getline < "addresslist")
         if ($0 ~ ARGV[1])
           print $0
}' $*
```

All processing is done in the **BEGIN** action.

Notice that any regular expression can be passed to **addr**; in particular, it is possible to retrieve by parts of an address or telephone number, as well as by name.

Example Applications

`awk` has been used in surprising ways. We have seen `awk` programs that implement database systems and a variety of compilers and assemblers, in addition to the more traditional tasks of information retrieval, data manipulation, and report generation. Invariably, the `awk` programs are significantly shorter than equivalent programs written in more conventional programming languages, such as Pascal or C. In this section, we will present a few more examples to illustrate some additional `awk` programs.

Generating Reports

`awk` is especially useful for producing reports that summarize and format information. Suppose we wish to produce a report from the file *countries* in which we list the continents alphabetically, and after each continent its countries in decreasing order of population:

```
Africa:
  Sudan      19
  Algeria    18

Asia:
  China      866
  India      637
  USSR       262

Australia:
  Australia  14

North America:
  USA        219
  Canada     24

South America:
  Brazil     116
  Argentina  26
```

As with many data processing tasks, it is much easier to produce this report in several stages. First, we create a list of continent-country-population triples, in which each field is separated by a colon. This can be done with the following program, `triples`, which uses an array `pop`, indexed by subscripts of the form 'continent:country' to store the popula-

tion of a given country. The **print** statement in the END section of the program creates the list of continent-country-population triples that are piped to the *sort* routine.

```
BEGIN { FS = "\t" }
      { pop[$4 ":" $1] += $3 }
END   { for (cc in pop)
        print cc ":" pop[cc] | "sort -t: +0 -1 +2nr" }
```

The arguments for *sort* deserve special mention. The **-t:** argument tells *sort* to use **:** as its field separator. The **+0 -1** arguments make the first field the primary sort key. In general, **+i -j** makes fields *i+1*, *i+2*, . . . , *j* the sort key. If **-j** is omitted, the fields from *i+1* to the end of the record are used. The **+2nr** argument makes the third field, numerically decreasing, the secondary sort key (*n* is for numeric, *r* for reverse order). Invoked on the file *countries*, this program produces as output:

```
Africa:Sudan:19
Africa:Algeria:18
Asia:China:866
Asia:India:637
Asia:USSR:262
Australia:Australia:14
North America:USA:219
North America:Canada:24
South America:Brazil:116
South America:Argentina:26
```

This output is in the right order but the wrong format. To transform the output into the desired form we run it through a second **awk** program, **format**.

```
BEGIN { FS = ":" }
      {
        if ($1 != prev) {
          print "\n" $1 ":"
          prev = $1
        }
        printf "\t%-10s %6d\n", $2, $3
      }
}
```

This is a control-break program that prints only the first occurrence of a continent name and formats the country-population lines associated with that continent in the desired manner. The command line

awk -f triples countries | awk -f format

Example Applications

gives us our desired report. As this example suggests, complex data transformation and formatting tasks can often be reduced to a few simple **awks** and **sorts**.

As an exercise, add to the population report subtotals for each continent and a grand total.

Additional Examples

Word Frequencies

Our first example illustrates associative arrays for counting. Suppose we want to count the number of times each word appears in the input, where a word equals any contiguous sequence of non-blank, non-tab characters. The following program prints the word frequencies, sorted in decreasing order.

```
    { for (w = 1; w <= NF; w++) count[$w]++ }
END { for (w in count) print count[w], w | "sort -nr" }
```

The first statement uses the array *count* to accumulate the number of times each word is used. Once the input has been read, the second for loop pipes the final count, along with each word, into the *sort* command.

Accumulation

Suppose we have two files, *deposits* and *withdrawals*, of records containing a name field and an amount field. For each name we want to print the net balance determined by subtracting the total withdrawals from the total deposits for each name. The net balance can be computed by the following program:

```
awk '
FILENAME == "deposits"    { balance[$1] += $2 }
FILENAME == "withdrawals" { balance[$1] -= $2 }
END                      { for (name in balance)
                          print name, balance[name]
                          } ' deposits withdrawals
```

The first statement uses the array *balance* to accumulate the total amount for each name in the file *deposits*. The second statement subtracts associated withdrawals from each total. If there are only withdrawals associated with a name, an entry for that name is created by the second statement. The END action prints each name with its net balance.

Random Choice

The following function prints (in order) k random elements from the first n elements of the array A . In the program, k is the number of entries that still need to be printed, and n is the number of elements yet to be examined. The decision of whether to print the i th element is determined by the test $\text{rand}() < k/n$.

```
function choose(A, k, n) {
    for (i = 1; n > 0; i++)
        if (rand() < k/n--) {
            print A[i]
            k--
        }
}
```

Shell Facility

The following **awk** program simulates (crudely) the history facility of the UNIX system shell. A line containing only `=` re-executes the last command executed. A line beginning with `= cmd` re-executes the last command whose invocation included the string `cmd`. Otherwise, the current line is executed.

```
$1 == "=" { if (NF == 1)
             system(x[NR] = x[NR-1])
           else
             for (i = NR-1; i > 0; i--)
                 if (x[i] ~ $2) {
                     system(x[NR] = x[i])
                     break
                 }
           next }
././      { system(x[NR] = $0) }
```

Example Applications

Form-Letter Generation

The following program generates form letters.

```
BEGIN {   FS = "|"
        while (getline <"form.letter")
            line[++n] = $0
    }
    {   for (i = 1; i <= n; i++) {
        s = line[i]
        for (j = 1; j <= NF; j++)
            gsub("\\$"j, $j, s)
        print s
    }
}
```

This program uses a template stored in a file called *form.letter*:

```
This is a form letter.
The first field is $1, the second $2, the third $3.
The third is $3, second is $2, and first is $1.
```

combined with replacement text of this form:

```
field 1|field 2|field 3
one|two|three
a|b|c
```

The BEGIN action stores the template in the array *template*; the remaining action cycles through the input data, using *gsub* to replace template fields of the form *\$n* with the corresponding data fields.

In all such examples, a prudent strategy is to start with a small version and expand it, trying out each aspect before moving on to the next.

awk Summary

The following section summarizes the functions and usage of **awk**.

Command Line

awk *program filenames*

awk -f *program-file filenames*

awk -Fs sets field separator to string *s*; **-Ft** sets separator to tab

Patterns

BEGIN

END

/regular expression/

relational expression

pattern && pattern

pattern || pattern

(pattern)

!pattern

pattern, pattern

Control Flow Statements

if (expr) statement [else statement]

if (subscript in array) statement [else statement]

while (expr) statement

for (expr; expr; expr) statement

for (var in array) statement

do statement while (expr)

break

continue

next

exit [*expr*]

return [*expr*]

Input-Output

close (<i>filename</i>)	close file
getline	set \$0 from next input record; set NF, NR, FNR
getline < <i>file</i>	set \$0 from next record of <i>file</i> ; set NF
getline <i>var</i>	set <i>var</i> from next input record; set NR, FNR
getline <i>var</i> < <i>file</i>	set <i>var</i> from next record of <i>file</i>
print	print current record
print <i>expr-list</i>	print expressions
print <i>expr-list</i> > <i>file</i>	print expressions on <i>file</i>
printf <i>fmt</i> , <i>expr-list</i>	format and print
printf <i>fmt</i> , <i>expr-list</i> > <i>file</i>	format and print on <i>file</i>
system (<i>cmd-line</i>)	execute command <i>cmd-line</i> , return status

In **print** and **printf** above, >> *file* appends to the *file*, and | *command* writes on a pipe. Similarly, *command* | **getline** pipes into **getline**. **getline** returns 0 on end of file, and -1 on error.

Functions

func *name*(*parameter list*) { *statement* }
function *name*(*parameter list*) { *statement* }
function-name(*expr*, *expr*, ...)

String Functions

gsub (<i>r</i> , <i>s</i> , <i>t</i>)	substitute string <i>s</i> for each substring matching regular expression <i>r</i> in string <i>t</i> , return number of substitutions; if <i>t</i> omitted, use \$0
index (<i>s</i> , <i>t</i>)	return index of string <i>t</i> in string <i>s</i> , or 0 if not present
length (<i>s</i>)	return length of string <i>s</i>
match (<i>s</i> , <i>r</i>)	return position in <i>s</i> where regular expression <i>r</i> occurs, or 0 if <i>r</i> is not present
split (<i>s</i> , <i>a</i> , <i>r</i>)	split string <i>s</i> into array <i>a</i> on regular expression <i>r</i> , return number of fields; if <i>r</i> omitted, FS is used in its place

sprintf (<i>fmt</i> , <i>expr-list</i>)	print <i>expr-list</i> according to <i>fmt</i> , return resulting string
sub (<i>r</i> , <i>s</i> , <i>t</i>)	like gsub except only the first matching substring is replaced
substr (<i>s</i> , <i>i</i> , <i>n</i>)	return <i>n</i> -char substring of <i>s</i> starting at <i>i</i> ; if <i>n</i> omitted, use rest of <i>s</i>

Arithmetic Functions

atan2 (<i>y</i> , <i>x</i>)	arctangent of <i>y</i> / <i>x</i> in radians
cos (<i>expr</i>)	cosine (angle in radians)
exp (<i>expr</i>)	exponential
int (<i>expr</i>)	truncate to integer
log (<i>expr</i>)	natural logarithm
rand ()	random number between 0 and 1
sin (<i>expr</i>)	sine (angle in radians)
sqrt (<i>expr</i>)	square root
srand (<i>expr</i>)	new seed for random number generator; use time of day if no <i>expr</i>

Operators (Increasing Precedence)

= += -= *= /= %= ^=	assignment
?:	conditional expression
	logical OR
&&	logical AND
~ !-	regular expression match, negated match
< <= > >= != ==	relationals
<i>blank</i>	string concatenation
+ -	add, subtract
* / %	multiply, divide, mod
+ - !	unary plus, unary minus, logical negation
^	exponentiation (** is a synonym)
++ --	increment, decrement (prefix and postfix)
\$	field

Regular Expressions (Increasing Precedence)

<code>c</code>	matches non-metacharacter <code>c</code>
<code>\c</code>	matches literal character <code>c</code>
<code>.</code>	matches any character but newline
<code>^</code>	matches beginning of line or string
<code>\$</code>	matches end of line or string
<code>[abc...]</code>	character class matches any of <code>abc...</code>
<code>[^abc...]</code>	negated class matches any but <code>abc...</code> and newline
<code>r1 r2</code>	matches either <code>r1</code> or <code>r2</code>
<code>r1r2</code>	concatenation: matches <code>r1</code> , then <code>r2</code>
<code>r+</code>	matches one or more <code>r</code> 's
<code>r*</code>	matches zero or more <code>r</code> 's
<code>r?</code>	matches zero or one <code>r</code> 's
<code>(r)</code>	grouping: matches <code>r</code>

Built-in Variables

ARGC	number of command-line arguments
ARGV	array of command-line arguments (0.. ARGC-1)
FILENAME	name of current input file
FNR	input record number in current file
FS	input field separator (default blank)
NF	number of fields in current input record
NR	input record number since beginning
OFMT	output format for numbers (default <code>%.6g</code>)
OFS	output field separator (default blank)
ORS	output record separator (default newline)
RS	input record separator (default newline)
RSTART	index of first character matched by <code>match()</code> ; 0 if no match
RLENGTH	length of string matched by <code>match()</code> ; -1 if no match
SUBSEP	separates multiple subscripts in array elements; default <code>"\034"</code>

Limits

Any particular implementation of `awk` enforces some limits. Here are typical values:

```

100 fields
2500 characters per input record
2500 characters per output record
1024 characters per individual field
1024 characters per printf string
400 characters maximum quoted string
400 characters in character class
15 open files
1 pipe
numbers are limited to what can be represented on the local
  machine, e.g., 1e-38..1e+38

```

Initialization, Comparison, and Type Coercion

Each variable and field can potentially be a string or a number or both at any time. When a variable is set by the assignment

```
var = expr
```

its type is set to that of the expression. (Assignment includes +=, -=, etc.) An arithmetic expression is of type number, a concatenation is of type string, and so on. If the assignment is a simple copy, as in

```
v1 = v2
```

then the type of v1 becomes that of v2.

In comparisons, if both operands are numeric, the comparison is made numerically. Otherwise, operands are coerced to string if necessary, and the comparison is made on strings. The type of any expression can be coerced to numeric by subterfuges such as

```
expr + 0
```

and to string by

```
expr ""
```

(that is, concatenation with a null string).

Uninitialized variables have the numeric value **0** and the string value "".

awk Summary

The type of a field is determined by context when possible; for example,

```
$1++
```

clearly implies that `$1` is to be numeric, and

```
$1 = $1 ", " $2
```

implies that `$1` and `$2` are both to be strings. Coercion is done as needed.

In contexts where types cannot be reliably determined, for example,

```
if ($1 == $2) ...
```

the type of each field is determined on input. All fields are strings; in addition, each field that contains only a number is also considered numeric.

Fields that are explicitly null have the string value `""`; they are not numeric. Non-existent fields (i.e., fields past `NF`) are treated this way, too.

As it is for fields, so it is for array elements created by `split()`.

Mentioning a variable in an expression causes it to exist, with the value `""` as described above. Thus, if `arr[i]` does not currently exist,

```
if (arr[i] == "") ...
```

causes it to exist with the value `""` so the `if` is satisfied. The special construction

```
if (i in arr) ...
```

determines if `arr[i]` exists without the side effect of creating it if it does not.

Chapter 12

Using the Stream Editor: sed

Introduction 12-1

Overall Operation 12-2

Addresses 12-4

Functions 12-6

Whole-Line Oriented Functions 12-6

Substitute Functions 12-8

Input-Output Functions 12-10

Multiple Input-Line Functions 12-12

Hold and Get Functions 12-12

Flow-of-Control Functions 12-13

Miscellaneous Functions 12-14

Introduction

This chapter describes the stream editor, **sed**, that allow you to perform large-scale, noninteractive editing tasks. The **sed** editor is useful if you must work with large files or run a complicated sequence of editing commands on a file or group of files.

12

Although you can perform many of the same tasks with **grep**, **sort**, and the variants of **diff**, you will find that **sed** offers an added facility for the processing of complicated changes to large files, or many files at once. **sed** is very handy for large batch editing jobs, but if you choose not to learn it, many of the same tasks can be performed with **ed** scripts.

The **sed** program is a noninteractive editor which is especially useful when the files to be edited are either too large, or the sequence of editing commands too complex, to be executed interactively. **sed** works on only a few lines of input at a time and does not use temporary files, so the only limit on the size of the files you can process is that both the input and output must be able to fit simultaneously on your disk. You can apply multiple “global” editing functions to your text in one pass. Since you can create complicated editing scripts and submit them to **sed** as a command file, you can save yourself considerable retyping and the possibility of making errors. You can also save and reuse **sed** command files which perform editing operations you need to repeat frequently.

Processing files with **sed** command files is more efficient than using **ed**, even if you prepare a prewritten script. Note, however, that **sed** lacks relative addressing because it processes a file one line at a time. Also, **sed** gives you no immediate verification that a command has altered your text in the way you actually intended. Check your output carefully.

The **sed** program is derived from **ed**, although there are considerable differences between the two, resulting from the different characteristics of interactive and batch operation. You will notice a striking resemblance in the class of regular expressions they recognize. The code for matching patterns is nearly identical for **ed** and **sed**.

Overall Operation

By default, **sed** copies the standard input to the standard output, performing one or more editing commands on each line before writing it to the output. Typically, you will need to specify the file or files you are processing, along with the name of the command file which contains your editing script, as in the following:

```
sed -f script filename
```

The flags are optional. The **-n** flag tells **sed** to copy only those lines specified by **-p** functions or **-p** flags after **-s** functions. The **-e** flag tells **sed** to take the next argument as an editing command, and the **-f** flag tells **sed** to take the next argument as a filename. (This file must contain editing commands, one to a line.)

The general format of a **sed** editing command is:

address1, address2 function arguments

In any command, one or both addresses may be omitted. A function is always required, but an argument is optional for some functions. Any number of blanks or tabs may separate the addresses from the function, and tab characters and spaces at the beginning of lines are ignored.

Three flags are recognized on the command line:

- n** Directs **sed** to copy only those lines specified by **p** functions or **p** flags after **s** functions.
- e** Indicates that the next argument is an editing command.
- f** Indicates that the next argument is the name of the file which contains editing commands, typed one to a line.

sed commands are applied one at a time, generally in the order they are encountered, unless you change this order with one of the “flow-of-control” functions discussed below. **sed** works in two phases, compiling the editing commands in the order they are given, then executing the commands one by one to each line of the input file.

The input to each command is the output of all preceding commands. Even if you change this default order of applying commands with one of the two flow-of-control commands, **t** and **b**, the input line to any command is still the output of any previously applied command.

You should also note that the range of pattern match is normally one line of input text. This range is called the “pattern space.” More than one line can be read into the pattern space by using the N command described below in “Multiple Input-Line Functions”.

The rest of this section discusses the principles of sed addressing, followed by a description of sed functions. All the examples here are based on the following lines from Samuel Taylor Coleridge’s poem, “Kubla Khan”:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

For example, the command:

```
2q
```

will quit after copying the first two lines of the input. Using the sample text, the result will be:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
```

Addresses

The following rules apply to addressing in `sed`. There are two ways to select the lines in the input file to which editing commands are to be applied: with line numbers or with “context addresses”. Context addresses correspond to regular expressions. The application of a group of commands can be controlled by one address or an address pair, by grouping the commands with curly braces (`{ }`). There may be 0, 1, or 2 addresses specified, depending on the command. The maximum number of addresses possible for each command is indicated.

A line number is a decimal integer. As each line is read from the input file, a line number counter is incremented. A line number address matches the input line, causing the internal counter to equal the address line number. The counter runs cumulatively through multiple input files. It is not reset when a new input file is opened. A special case is the dollar sign character (`$`) which matches the last line of the last input file.

Context addresses are enclosed in slashes (`/`). They include all the regular expressions common to both `ed` and `sed`:

1. An ordinary character is a regular expression and matches itself.
2. A caret (`^`) at the beginning of a regular expression matches the null character at the beginning of a line.
3. A dollar sign (`$`) at the end of a regular expression matches the null character at the end of a line.
4. The characters `\n` match an embedded newline character, but not the newline at the end of a pattern space.
5. A period (`.`) matches any character except the terminal newline of the pattern space.
6. A regular expression followed by a star (`*`) matches any number, including 0, of adjacent occurrences of regular expressions.
7. A string of characters in square brackets (`[]`) matches any character in the string, and no others. If, however, the first character of the string is a caret (`^`), the regular expression matches any character except the characters in the string and the terminal newline of the pattern space.

8. A concatenation of regular expressions is one that matches a particular concatenation of strings.
9. A regular expression between the sequences “\(' and ‘\)’” is identical in effect to itself, but has side-effects with the `s` command. (Note the following specification.)
10. The expression `\d` means the same string of characters matched by an expression enclosed in `\(` and `\)` earlier in the same pattern. Here “`d`” is a single digit; the string specified is that beginning with the “`dth`” occurrence of `\(`, counting from the left. For example, the expression `\(.*\)\1` matches a line beginning with two repeated occurrences of the same string.
11. The null regular expression standing alone is equivalent to the last regular expression compiled.

12

For a context address to “match” the input, the whole pattern within the address must match some portion of the pattern space. If you want to use one of the special characters literally, that is, to match an occurrence of itself in the input file, precede the character with a backslash (`\`) in the command.

Each `sed` command can have 0, 1, or 2 addresses. The maximum number of allowed addresses is included. A command with no addresses specified is applied to every line in the input. If a command has one address, it is applied to all lines which match that address. On the other hand, if two addresses are specified, the command is applied to the first line which matches the first address, and to all subsequent lines until and including the first subsequent line which matches the second address. An attempt is made on subsequent lines to again match the first address, and the process is repeated. Two addresses are separated by a comma. Here are some examples:

<code>/an/</code>	Matches lines 1, 3, 4 in our sample text
<code>/an.*an/</code>	Matches line 1
<code>/^an/</code>	Matches no lines
<code>/./</code>	Matches all lines
<code>/r*an/</code>	Matches lines 1,3, 4 (number = zero!)

Functions

All `sed` functions are named by a single character. They are of the following types:

- Whole-line oriented functions which add, delete, and change whole text lines.
- Substitute functions which search and substitute regular expressions within a line.
- Input-output functions which read and write lines and/or files.
- Multiple input-line functions which match patterns that extend across line boundaries.
- Hold and get functions which save and retrieve input text for later use.
- Flow-of-control functions which control the order of application of functions.
- Miscellaneous functions.

Whole-Line Oriented Functions

- d** Deletes from the file all lines matched by its addresses. No further commands will be executed on a deleted line. As soon as the `d` function is executed, a newline is read from the input, and the list of editing commands is restarted from the beginning on the newline. The maximum number of addresses is two.
- n** Reads and replaces the current line from the input, writing the current line to the output if specified. The list of editing commands is continued following the `n` command. The maximum number of addresses is two.
- a** Causes the text to be written to the output after the line matched by its address. The `a` command is inherently multiline; The `a` command must appear at the end of a line. The text may contain any number of lines. The

interior newlines must be hidden by a backslash character (\) immediately preceding each newline. The text argument is terminated by the first unhidden newline, the first one not immediately preceded by backslash. Once an a function is successfully executed, the text will be written to the output regardless of what later commands do to the line which triggered it, even if the line is subsequently deleted. The text is not scanned for address matches, and no editing commands are attempted on it, nor does it cause any change in the line number counter. Only one address is possible.

- i When followed by a text argument it is the same as the a function, except that the text is written to the output before the matched line. It has only one possible address.

- c The c function deletes the lines selected by its addresses, and replaces them with the lines in the text. Like the a and i commands, c must be followed by a newline hidden with a backslash; interior newlines in the text must be hidden by backslashes. The c command may have two addresses, and therefore select a range of lines. If it does, all the lines in the range are deleted, but only one copy of the text is written to the output, not one copy per line deleted. As in the case of a and i, the text is not scanned for address matches, and no editing commands are attempted on it. It does not change the line number counter. After a line has been deleted by a c function, no further commands are attempted on it. If text is appended after a line by a or r functions, and the line is subsequently changed, the text inserted by the c function will be placed before the text of the a or r functions.

Note that when you insert text in the output with these functions, leading blanks and tabs will disappear in all sed commands. To get leading blanks and tabs into the output, precede the first desired blank or tab by a backslash; the backslash will not appear in the output.

Functions

For example, the list of editing commands:

```
n
a\
XXXX
d
```

applied to our standard input, produces:

```
In Xanadu did Kubla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
```

In this particular case, the same effect would be produced by either of the two following command lists:

```
n
i\
XXXX
d
```

or:

```
n
c\
XXXX
```

Substitute Functions

The substitute function(s) changes parts of lines selected by a context search within the line, as in:

(2)s *pattern replacement flags* substitute

The s function replaces part of a line selected by the designated pattern with the replacement pattern. The pattern argument contains a **pattern**, exactly like the patterns in addresses. The only difference between a pattern and a context address is that a pattern argument may be delimited by any character other than space or newline. By default, only the first string matched by the pattern is replaced, except when the -g option is used.

The replacement argument begins immediately after the second delimiting character of the pattern, and must be followed immediately by another instance of the delimiting character. The replacement is not a pattern,

and the characters which are special in patterns do not have special meaning in replacement. Instead, the following characters are special:

- .
- Is replaced by the string matched by the pattern.
- `\d` *d* is a single digit which is replaced by the *d*th substring matched by parts of the pattern enclosed in `\(` and `\)`. If nested substrings occur in the pattern, the *d*th substring is determined by counting opening delimiters .

As in patterns, special characters may be made literal by preceding them with a backslash (`\`).

A flag argument may contain the following:

- g** Substitutes the replacement for all nonoverlapping instances of the pattern in the line. After a successful substitution, the scan for the next instance of the pattern begins just after the end of the inserted characters; characters put into the line from the replacement are not rescanned.
- p** Prints the line if a successful replacement was done. The **p** flag causes the line to be written to the output only if a substitution was actually made by the **s** function. Notice that if several **s** functions, each followed by a **p** flag, successfully substitute in the same input line, multiple copies of the line will be written to the output: one for each successful substitution.
- w file** Writes the line to a file if a successful replacement was done. The **-w** option causes lines which are actually substituted by the **s** function to be written to the named file. If the filename existed before **sed** is run, it is overwritten; if not, the file is created. A single space must separate **-w** and the filename. The possibilities of multiple, somewhat different copies of one input line being written are the same as for the **-p** option. A combined maximum of ten different filenames may be mentioned after **w** flags and **w** functions.

Here are some examples. (Only the lines affected by the changes are shown for the sake of clarity. In reality, even unchanged lines would be passed through and printed.) When applied to our standard input, the following command:

```
s/to/by/w changes
```

Functions

produces, on the standard output:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless by man
Down by a sunless sea.
```

and on the file *changes*:

```
Through caverns measureless by man
Down by a sunless sea.
```

The command:

```
s/[.,;?:]/*P&*/gp
```

produces:

```
A stately pleasure dome decree*P:*
Where Alph*P,* the sacred river*P,* ran
Down to a sunless sea*P.*
```

With the *g* flag, the command:

```
/X/s/an/AN/p
```

produces:

```
In XANadu did Kubla Khan
```

and the command:

```
/X/s/an/AN/gp
```

produces:

```
In XANadu did Kubla KhAN
```

Input-Output Functions

- p** The print function writes the addressed lines to the standard output file at the time the *p* function is encountered, regardless of what succeeding editing commands may do to the lines. The maximum number of possible addresses is two.

- w** The write function writes the addressed lines to *filename*. If the file previously existed, it is overwritten; if not, it is created. The lines are written exactly as they exist when the write function is encountered for each line, regardless of what subsequent editing commands may do to them. Exactly one space must separate the w command and the filename. The combined number of write functions and w flags may not exceed 10.
- r** The read function reads the contents of the named file, and appends them after the line matched by the address. The file is read and appended regardless of what subsequent editing commands do to the line which matched its address. If **r** and **a** functions are executed on the same line, the text from the **a** functions and the **r** functions is written to the output in the order that the functions are executed. Exactly one space must separate the **r** and the filename. One address is possible. If a file mentioned by an **r** function cannot be opened, it is considered a null file rather than an error, and no diagnostic is given.

Note that since there is a limit to the number of files that can be opened simultaneously, be sure that no more than ten files are mentioned in functions or flags; that number is reduced by one if any **r** functions are present. Only one read file is open at one time.

Here are some examples. Assume that the file *notel* has the following contents:

```
Note: Kubla Khan (more properly Kublai Khan;
1216-1294) was the grandson and most eminent
successor of Genghiz (Chingiz) Khan, and
founder of the Mongol dynasty in China.
```

The command:

```
/Kubla/r notel
```

produces:

Functions

In Xanadu did Kubla Khan

Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was the grandson and most eminent successor of Genghiz (Chingiz) Khan, and founder of the Mongol dynasty in China.

A stately pleasure dome decree:

Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.

Multiple Input-Line Functions

Three functions, all spelled with upper-case letters, deal specially with pattern spaces containing embedded newlines. They are intended principally to provide pattern matches across lines in the input.

- N** Appends the next input line to the current line in the pattern space; the two input lines are separated by an embedded newline. Pattern matches may extend across the embedded newline(s). There is a maximum of two addresses.
- D** Deletes up to and including the first newline character in the current pattern space. If the pattern space becomes empty (the only newline was the terminal newline), another line is read from the input. In any case, begin the list of editing commands over again. The maximum number of addresses is two.
- P** Prints up to and including the first newline in the pattern space. The maximum number of addresses is two.

The **P** and **D** functions: these functions are equivalent to their lowercase counterparts if there are no embedded newlines in the pattern space.

Hold and Get Functions

These functions save and retrieve part of the input for possible later use:

- h** The **h** function copies the contents of the pattern space into a holding area, destroying any previous contents of the holding area. The maximum number of addresses is two.

- H The **H** function appends the contents of the pattern space to the contents of the holding area. The former and new contents are separated by a newline.
- g The **g** function copies the contents of the holding area into the pattern space, destroying the previous contents of the pattern space.
- G The **G** function appends the contents of the holding area to the contents of the pattern space. The former and new contents are separated by a newline. The maximum number of addresses is two.
- x The exchange command interchanges the contents of the pattern space and the holding area. The maximum number of addresses is two.

For example, the commands:

```
lh
ls/ did.*//
lx
G
s/\n/  :/
```

applied to our standard example, produce:

```
In Xanadu did Kubla Khan  :In Xanadu
A stately pleasure dome decree:  :In Xanadu
Where Alph, the sacred river, ran  :In Xanadu
Through caverns measureless to man  :In Xanadu
Down to a sunless sea.  :In Xanadu
```

Flow-of-Control Functions

These functions do no editing on the input lines, but control the application of functions to the lines selected by the address part.

- ! This command causes the next command written on the same line to be applied to only those input lines not selected by the address part. There are two possible addresses.

Functions

- {** This command causes the next set of commands to be applied or not applied as a block to the input lines selected by the addresses of the grouping command. The first of the commands under control of the grouping command may appear on the same line as the **{** or on the next line. The group of commands is terminated by a matching **}** on a line by itself. Groups can be nested and may have two addresses.
- :label** The label function marks a place in the list of editing commands which may be referred to by **b** and **t** functions. The *label* may be any sequence of eight or fewer characters; if two different colon functions have identical labels, an error message will be generated, and no execution attempted.
- label* The branch function causes the sequence of editing commands being applied to the current input line to be restarted immediately after encountering a colon function with the same label. If no colon function with the same label can be found after all the editing commands have been compiled, an error message is produced, and no execution is attempted. A **b** function with no label is interpreted as a branch to the end of the list of editing commands. Whatever should be done with the current input line is done, and another input line is read; the list of editing commands is restarted from the beginning on the new line. Two addresses are possible.
- label* The **t** function tests whether any successful substitutions have been made on the current input line. If so, it branches to the label; if not, it does nothing. The flag which indicates that a successful substitution has been executed is reset either by reading a new input line, or by executing a **t** function.

Miscellaneous Functions

There are two other functions of **sed** not discussed above.

- =** The **=** function writes to the standard output the number of the line matched by its address. One address is possible.

- q** The **q** function causes the current line to be written to the output (if it should be), any appended or read text to be written, and execution to be terminated. One address is possible.



Chapter 13

Using the Job Scheduling Commands: at, cron and batch

Introduction 13-1

Automatic Program Execution with cron 13-2

Delaying Program Execution with batch and at 13-4

Introduction

This chapter explains how to use the **cron(C)**, **at(C)**, and **batch(C)** programs to schedule or delay the execution of programs (jobs). Each of these utilities is subject to authorization by the system administrator, who can permit or deny their use by ordinary users.

The role of the job scheduling programs is as follows:

- | | |
|--------------|--|
| cron | Executes programs repeatedly at a specified time. |
| at | Delays the execution of programs until a time specified by the user. |
| batch | Delays the execution of programs until the system load is low (as determined by the system). |

13

This chapter explains how the user or system administrator can use each of the job scheduling programs to automate regular operations or delay execution of programs that would otherwise slow down the system during peak usage.

Automatic Program Execution with cron

Altos UNIX System V systems allow you to have programs run automatically at specified times. This is done with the **cron** program. The **cron** program and, more specifically, the **crontab** command allow you to run programs during off-hours such as

- file system administration
- long-running, user-written shell procedures
- cleanup procedures

Any task that needs to be done repeatedly at a specified time is a candidate for your *cron* file located in the */usr/spool/cron/crontabs* directory. You can use the **crontab** command to establish the entries you want.

The **crontab** command is used as follows:

```
crontab file  
crontab -r  
crontab -l
```

The **crontab** command copies the specified *file* or standard input if no file is specified into a directory that holds all users' crontabs. The **-r** option removes a user's crontab from the *crontab* directory. The **-l** option will list the *crontab* file for the invoking user. See the **crontab(C)** command in the for additional information.

Each line in the *crontab* file defines one procedure. The line entry format looks like the following:

```
minute hour day month day-of-week command
```

Each field is defined as follows:

```
minute (0-59),  
hour (0-23),  
day (1-31),  
month (1-12),  
day-of-week (0-6 with 0=Sunday)  
command (the command to be executed at the time specified)
```

The following rules apply to the first five fields:

- Two numbers separated by a hyphen indicate a range of numbers between the two specified numbers.
- A list of numbers separated by commas indicates only the numbers listed will be used.
- An asterisk specifies all legal values.

For example, `0 0 1,14 * 2` indicates a command will be run on the first and fourteenth of each month, as well as on every Tuesday. If a percent sign (%) is placed in the command field (sixth field), the operating system will translate it as a new-line character. Only the first line of a command field (character string up to the percent sign) is executed by the shell. Any other lines are made available to the command as standard input.

13

For example, let a file called *anyfile* contain the following **cron** entry:

```
0 0 1 * * mailx $LOGNAME % Subject: Call Mom! % now
```

When the command line **crontab anyfile** is executed, the user whose login is **\$LOGNAME** will get a reminder mail message with “Call Mom!” as the subject the first of every month.

Delaying Program Execution with batch and at

The **batch** and **at** commands allow you to specify a command or sequence of commands to be run at a later time. With the **batch** command, the system determines when the commands run; with the **at** command, you determine when the commands run. Both commands expect input from standard input (the terminal); the list of commands entered as input from the terminal must be ended by pressing **<CTL>d**.

The **batch** command is useful if you are running a process or shell program that uses a large amount of system time. The **batch** command submits a batch job (containing the commands to be executed) to the system. The job is put in a queue and runs when the system load falls to an acceptable level. This frees the system to respond rapidly to other input and is a courtesy to other users. Note that if the system load is light, the submitted **batch** job is executed immediately.

The general format for **batch** is

```
batch  
first command  
.  
.  
.  
last command  
<CTL>d
```

If there is only one command to be run with **batch**, you can enter it as follows:

```
batch command_line  
<CTL>d
```

The next example uses **batch** to execute the **grep** command at a convenient time. Here **grep** searches all files in the current directory for “dollar” and redirects the output to the file **dol.file**:

```
$ batch grep dollar * > dol-file  
<CTL>d  
job 155223141.b at Sun Dec 7 11:14:54 1989  
$
```

After you submit a job with **batch**, the system responds with a job number, date, and time. This job number is not the same as the process number that the system generates when you run a command in the background.

Figure 13-1 summarizes the syntax and capabilities of the **batch** Command.

Figure 13-1 Summary of the **batch** Command

batch - executes commands at a later time		
<i>command</i>	<i>options</i>	<i>input</i>
batch	none	<i>command_lines</i>
Description:	batch submits a batch job, which is placed in a queue and executed when the load on the system falls to an acceptable level.	
Remarks:	The list of commands must end with a <CTL>d.	

The **at** command allows you to specify an exact time to execute the commands. The general format for the **at** command is

```

at time
first command
.
.
.
last command
<CTL>d
    
```

The *time* argument consists of the time of day and, if the date is not today, the date.

The following example shows how to use the **at** command to mail a happy birthday banner to login **emily** on her birthday:

Delaying Program Execution with batch and at

```
$ at 8:15am Feb 27
banner happy birthday | mail emily
(CTL)d
job 453400603.a at Thurs Feb 27 08:15:00 1986
$
```

Notice that the **at** command, like the **batch** command, responds with the job number, date, and time.

If you decide you do not want to execute the commands currently waiting in a **batch** or **at** job queue, you can erase those jobs by using the **-r** option of the **at** command with the job number. The general format is

```
at -r jobnumber
```

Try erasing the previous **at** job for the happy birthday banner. Enter:

```
at -r 453400603.a
```

If you have forgotten the job number, the **at -l** command will give you a list of the current jobs in the **batch** or **at** queue, as the following screen example shows:

```
$ at -l
user = mylogin 168302040.a at Sat Nov 29 13:00:00 1988
user = mylogin 453400603.a at Fri Feb 27 08:15:00 1989
$
```

Notice that the system displays the job number and the time the job will run.

Using the **at** command, mail yourself the file **memo** at noon to tell you it is lunch time. Then try the **at** command with the **-l** option:

```
$ at 12:00pm
mail mylogin < memo
(CTL)d
job 263131754.a at Jun 30 12:00:00 1989
$
$ at -l
user = mylogin 8263131754.a at Jun 30 12:00:00 1989
$
```

Figure 13-2 summarizes the syntax and capabilities of the `at` command.

Figure 13-2 Summary of the `at` Command

at - executes commands at a specified time		
<i>command</i>	<i>options</i>	<i>arguments</i>
at	-r -l	<i>time (date)</i> <i>jobnumber</i>
Description:	<p><code>at</code> executes commands at the time specified. You can use between one and four digits, and am or pm to show the time. To specify the date, give a month name followed by the number for the day. You do not need to enter a date if you want your job to run the same day. See the <code>at(C)</code> manual page in the <i>User's Reference</i> for other default times.</p>	
Options:	<p>The <code>-r</code> option with the job number removes previously scheduled jobs.</p> <p>The <code>-l</code> option (no arguments) reports the job number and status of all scheduled <code>at</code> and <code>batch</code> jobs.</p>	
Remarks:	<p>Examples of how to specify times and dates with the <code>at</code> command are as follows:</p> <p style="text-align: center;">at 08:15am Feb 27 at 5:14pm Sept 24</p>	

Chapter 14

Using DOS Accessing Utilities

Introduction 14-1

Accessing DOS Files with the dos(C) Utilities 14-2
 Copying Groups of Files 14-3

Using Mounted DOS Filesystems 14-5
 Mounting DOS Filesystem 14-5
 File and Directory Arguments 14-6
 User Configurable Default File 14-6
 Appearance of DOS Files 14-7
 Newline Conversions with DOS Utilities 14-8
 Other Restrictions 14-8



Introduction

DOS tools are provided a help you bridge between the two operating systems. These tools are an extension of the features available on UNIX systems. These programs allow you, while working on your Altos UNIX System V computer system, to access DOS files and directories which reside in a non-active DOS partition. If your system administrator permits, you can also gain access DOS files by mounting a DOS filesystem and using them directly. This chapter discusses both methods.

Accessing DOS Files with the dos(C) Utilities

The following is a list of the **dos(C)** commands and their functions:

- | | |
|------------------|---|
| doscat | Copies one or more DOS files to the standard output. By default, the standard output is the terminal screen. If more than one file is specified, the files are displayed concatenated together. doscat functions like the UNIX cat(C) command. The following is an example of its usage:
doscat/dev/fd0:/john/memos |
| doscp | Copies files between DOS and UNIX environments. The first file specified is copied to the second file. Example of usage:
doscp /mary/list /dev/fd0:/budget/list |
| dosdir | Lists DOS files in the standard DOS style directory format. Example of usage:
dosdir /dev/fd0:/john |
| dosformat | Creates a DOS 2.0 formatted diskette. Example of usage:
dosformat /dev/fd0 |
| dosls | Lists DOS files and directories in a UNIX format. Example of usage:
dosls /dev/fd0:/john |
| dosrm | Removes files from a DOS disk. Example of usage:
dosrm /dev/fd0:/john/memos |
| dosmkdir | Creates a directory on a DOS disk. Example of usage:
dosmkdir /dev/fd0:/john/memos |

dosrmdir Deletes directories from a DOS disk. Example of usage:

```
dosrmdir /dev/fd0:/john/memos
```

Note that you must have a bootable, although not active, DOS partition on the hard disk or a DOS floppy in order to use these UNIX commands. For example, you can only transfer a file from a UNIX partition on hard disk to a DOS floppy if either the DOS floppy is bootable or there is also a DOS partition on the hard disk. For more information about the DOS accessing utilities, refer to the *User's Reference*.

You may also be able to use the UNIX **dd(C)** and **diskcp(C)** commands to copy and compare DOS floppies. The UNIX **dtype(C)** command tells you what type of floppies you have (various DOS and UNIX floppies).

Also, the file */etc/default/msdos* describes which DOS filesystems (e.g. A:, B:, C: ...) correspond to which Altos UNIX System V devices.

Note

You cannot execute (run) DOS programs or applications under Altos UNIX System V.

If you have the Development System, with the *cmerge* compiler, you can create and compile programs that can be run under DOS operating systems. Refer to the *DOS-OS/2 Development Guide* for more information. Also, see the DOS section in the *Programmer's Reference*.

Copying Groups of Files

The **doscp** command does not allow the use of wildcards, so it is only possible to copy one file at a time. To work around this restriction so that you can copy groups of files to or from a DOS diskette or partition, you must create the following shell scripts:

DOS file format to UNIX file format transfer script:

Accessing DOS Files with the dos(C) Utilities

```
: fromdos: copy a batch of files from DOS to UNIX format
if [ "$1" = "" ]
then
    echo "Usage: $0 disk:[/dospath/directory]"
    exit 1
fi
dosdir=$1
names=`dosls $dosdir`
for i in $names
do
    doscp "$dosdir/$i" `echo $i | tr "[A-Z]" "[a-z]"`
done
```

UNIX file format to DOS file format transfer script:

```
: todos: copy a batch of files from UNIX format to DOS
if [ $# -lt 2 ]
then
    echo "Usage: $0 file [file ...] disk:[/dospath/directory]"
    exit 1
fi
files="$1"
while [ "$2" != "" ]
do
    files="$files $1"
    shift
done
dosdir=$1
for i in $files
do
    doscp $i $dosdir
done
```

Both of these scripts should be created as executable files under */usr/bin* and should be given appropriate names such as “*fromdos*” or “*todos*”. Use the **chmod(C)** command to make the files executable. For example:

```
chmod 755 /usr/bin/fromdos
```

gives read and execute permissions for the file *fromdos* to all users. Once these permissions are issued, you can use the filename as a substitute for **doscp** on your command line, as in the following example:

```
fromdos /dev/fd0:/john
```

Using Mounted DOS Filesystems

In addition to the DOS utilities provided with to manipulate DOS files, it is also possible to mount a DOS filesystem and access its files directly while still operating from the UNIX partition.

This means that DOS files can be edited or examined in place, without first copying them into the UNIX filesystem. The major restriction is that DOS files and applications cannot be executed under this arrangement; this requires use of VP/ix (if operating from the UNIX partition) or booting of the DOS partition. However, data files and text files can be examined, copied or edited.

The operating system deals with DOS filesystems by superimposing certain qualities of UNIX filesystems over the DOS filesystem without changing the actual files. UNIX filesystems are highly structured and operate in a multiuser environment. In order to make DOS files readily accessible, access permissions and file ownership are superimposed on the DOS filesystem when mounted.

14

Mounting DOS Filesystem

Only **root** can mount a filesystem. Access by users is governed by the permissions and ownership that **root** places on the DOS filesystem. The system administrator must either mount the DOS filesystem or set up the system so that users can use the **mnt(C)** command.

Example: Mounting a Floppy Disk

For example, if the system administrator permits it, you could mount DOS floppy disks, as in the following example using the 96tpi floppy mounted on */mnt*:

```
mnt -f DOS /dev/fd096 /mnt
```

Because of the limitations discussed earlier, DOS does not recognize permissions or ownership. When mounted from the UNIX partition, the DOS files behave as follows:

- The permissions and ownership of the filesystem are governed by the mount point. For example, if **root** creates a mount point */X* with permissions of *777*, all users can read or write the contents of

Using Mounted DOS Filesystems

the filesystem. If the mount point is owned by **root**, all files within the DOS filesystem and any created by other users are all owned by **root**.

- The permissions for regular files will be either 0777 for readable/writable files or 0555 for read only files. This preserves the consistency of the DOS filesystem. If a user can access the filesystem, the user will be limited by the permissions available under the DOS directory structure. This permission is read-only or read-write. When a file is created, the permissions are based on the **umask** of the creator. For example, assume the user's **umask** is 022, which generates files with permissions of 777. Here are further examples.

File and Directory Arguments

The file and directory arguments for DOS files take the form:

device:name

where *device* is a UNIX pathname for the special device file containing the DOS diskette or DOS partition, and *name* is a pathname to a DOS file or directory. For example,

/dev/fd0:/john/memos

indicates that the file *memos* is in the directory */john*, and that both are in the device file */dev/fd0* (the UNIX special device file for the primary floppy drive). Arguments without *device:* are assumed to be UNIX files.

User Configurable Default File

For convenience, the user configurable default file */etc/default/msdos* defines DOS drive names that you can use in place of UNIX special device file pathnames. These are short forms that the system administrator can set up for DOS filesystems using the A:, B:, C: convention instead of, for example, */dev/fd096ds15*.

Using Mounted DOS Filesystems

```
dtox KOMMIE.BAS > filename  
xtod KOMMIE.BAS > filename
```

Newline Conversions with DOS Utilities

When the `doscat(C)` and `doscp(C)` commands transfer DOS format text files to UNIX format, they automatically strip the `^M` character. When text files are transferred to DOS, the commands insert a `^M` before each linefeed character. Under some circumstances, the automatic newline conversions do not occur. The `-m` option ensures that the newline conversion is carried out. The `-r` option overrides the automatic conversion and forces the command to perform a true byte copy regardless of file type.

Note

All DOS utilities leave temporary files in `/tmp`, regardless of whether or not the utility executed successfully. These files are removed at the next reboot.

Other Restrictions

There are additional restrictions that must be observed.

File Names

The rules for file names and their conversion follows the guidelines found in the `dos(C)` manual page in the *User's Reference*. All DOS filenames have a maximum of eight characters, plus a three-character extension. For example, if you attempt to create a file named *rumplestiltski* within the DOS filesystem, it will look like this:

RUMPLEST

In addition, the standard DOS restrictions on illegal characters apply. However, wildcards can be used just as they can with UNIX filesystems.

Modification Times

When accessed from the UNIX partition, the creation, modification, and access times of DOS files are always identical and use GMT, or Greenwich Mean Time. (This is because UNIX systems use GMT internally and convert it for the user.) This means that files created in the DOS filesystem will not have consistent times across the operating systems.

UNIX Backup Utilities

The UNIX **backup(C)** utility cannot be used to make backups of a mounted DOS filesystem. DOS utilities and other copy programs like **tar(C)** will work as expected.

For more information, including more technical aspects of DOS usage, refer to the **dos(C)** page in the *User's Reference*.



Index

Special Characters

{ } command. *See* Braces command
({ })
: command. *See* Colon (:), command
. command. *See* Dot (.), command
! command. *See* escape command (!)
/ command. *See* vi, slash (/)
\$# variable, argument recording 7-16
\$! variable, background process number
7-17
\$? variable, command exit status 7-16
\$- variable, execution flags 7-17
\$\$ variable, process number 7-16
O command. *See* vi

A

A command
 append at end of line 2-24
a command
 appending text 2-24
 ed use. *See* ed
 mail 4-21, 4-22
 vi use. *See* vi
A Handful of Useful One-liners 11-9
-a operator 7-48
abbr command 2-62
Accumulation 11-50
actions
 awk 11-18
Actions 11-18
Addition. *See* bc
Additional Examples 11-50
addr
 awk 11-46, 11-47, 11-46, 11-47
Alias
 C-shell 8-10
 Korn shell 9-15
allexport option 9-17
Ampersand (&)
 See also And-if operator (&&)
 background process 7-26, 7-69, 9-10
 command list 7-26
 ed use. *See* ed
Ampersand (&) (*continued*)
 INTERRUPT and QUIT immunity
 7-26
 jobs to other computers 7-26
 metacharacter. *See* ed
 off-line printing 7-26
 use restraint 7-27
And-if operator (&&)
 command list 7-26
 described 7-27
 designated 7-69
Append
 See also Insert
 ed procedure. *See* ed
 output append symbol. *See* Output
 vi procedure 2-24
Argument
 filename 7-3
 list, creating 7-3
 mail commands 4-7
 number checking, \$# variable 7-16
 processing 7-23
 redirection argument, location 7-9
 shell, argument passing 7-23
 substitution sequence 7-24
 test command argument 7-48
Arguments
 command line 11-43
Arithmetic
 expr command effect 7-49, 11-18
Arithmetic functions
 awk 11-55
Arithmetic Functions 11-55
Arithmetic. *See* bc
arrays
 awk 11-30
Arrays 11-30
askcc option. *See* mail
Asterisk (*)
 bc
 comment convention 6-15, 6-16
 multiplication operator symbol 6-3
 , 6-5
 directory name, not used in 7-4
 mail
 message saved, header notation
 4-12, 4-6
 mail, all messages, symbol 4-8

Index

Asterisk (*) (*continued*)

- metacharacter 7-4, 7-70
- pattern matching 7-4
- special shell variable 7-24

Authorizations

- chmodsgid 10-7
- printerstat 10-8
- printqueue 10-8
- queryspace 10-8
- secondary 10-8
- subsystem 10-7

auths command 10-8

auto command, bc 6-21

awk

- actions 11-18
- addr 11-46, 11-47
- arithmetic 11-18
- arithmetic functions 11-55
- arrays 11-30
- basic 11-2
- BEGIN and END patterns 11-11
- built-in variables 11-18, 11-56, 11-8
- close statement 11-38
- command line 11-53
- command line argument 11-43
- comparison 11-57
- control flow statements 11-27, 11-53
- cooperation with shell 11-45
- error messages 11-10
- example applications 11-48
- field variable 11-25
- fields 11-4
- Files and Pipes 11-39
- formatted printing 11-6
- form-letter generation 11-52
- functions 11-54, 11-9
- generating reports 11-48
- getline function 11-40, 11-42
- gsub 11-22
- index 11-23
- initialization 11-57
- initialization, comparison, and type coercion 11-57
- input 11-39
- input separators 11-39
- input-output 11-54
- introduction 11-1
- lexical conventions 11-33
- limits 11-56
- match 11-23
- multi-line records 11-40
- operators 11-55
- output 11-34
- output into files 11-37

awk (*continued*)

- output into pipes 11-37
 - output separators 11-34
 - pattern combinations 11-16
 - pattern ranges 11-17
 - patterns 11-53, 11-11, 11-6
 - print statement 11-34
 - printf 11-35, 11-36, 11-6
 - printing 11-4
 - program structure 11-2
 - random choice 11-51
 - regular expressions 11-13
 - relational expressions 11-12
 - shell 11-45
 - shell facility 11-51
 - simple actions 11-8
 - sprintf 11-24
 - statements 11-27
 - string functions 11-21
 - strings 11-21
 - sub 11-22
 - substr 11-24
 - system 11-45
 - type coercion 11-57
 - usage 11-3
 - useful one-liners 11-9
 - user-defined functions 11-32
 - user-defined variables 11-8, 11-53
- awk Summary 11-53

B

b command. *See* vi

Background

job

C-shell use. *See* C-shell

process

#! variable 7-17

ampersand (&) operator 7-26, 7-69

dial-up line

Ctrl-d effect 7-26

nohup command 7-26

INTERUPT immunity 7-26

QUIT immunity 7-26

use restraint 7-27

Backslash (\)

bc

comment convention 6-15, 6-16

line continuation notation 6-7

C-shell use. *See* C-shell

ed use. *See* ed

line continuation notation 7-62

- Backslash (\) (*continued*)
 - metacharacter escape 7-4
 - quoting 7-70
- BACKSPACE key
 - bc 6-2
 - mail 4-9
- Basic
 - awk 11-2
- Basic awk 11-2
- bc
 - addition operator
 - evaluation order 6-17
 - left to right binding 6-5
 - scale 6-20 , 6-8
 - symbol (+) 6-5
 - additive operator
 - See also* specific operator
 - left to right binding 6-20
 - alphabetic register 6-3
 - arctan function
 - availability 6-1
 - loading procedure 6-15
 - array
 - auto array 6-21
 - characteristics 6-16
 - identifier 6-16 , 6-22
 - name 6-11
 - named expression 6-17
 - one-dimensional 6-11
 - assignment
 - operator
 - designated, use 6-20
 - evaluation order 6-17
 - positioning effect 6-6
 - symbol (=) 6-6
 - statement 6-14
 - asterisk (*)
 - comment convention 6-15 , 6-16
 - multiplication operator symbol 6-3 , 6-5
 - auto
 - command 6-21
 - keyword 6-16
 - statement
 - built-in statement 6-22
 - backslash (\)
 - comment convention 6-15 , 6-16
 - line continuation notation 6-7
 - BACKSPACE key 6-2
 - bases 6-6
 - bc command
 - file, reading and executing 6-15
 - invoking 6-2
 - bc -l command 6-15
 - bc (*continued*)
 - Bessel function
 - availability 6-1
 - loading procedure 6-15
 - braces ({})
 - compound statement enclosure 6-22
 - function body enclosure 6-9
 - brackets ([])
 - array identifier 6-16
 - auto array 6-21
 - subscripted variable 6-11
 - break, keyword 6-16
 - break statement
 - built-in statement 6-22
 - built-in statement 6-22
 - caret (^), exponentiation operator
 - symbol 6-5
 - comment convention 6-15 , 6-16
 - compound statement 6-22
 - constant
 - defined 6-17 , 6-18
 - construction
 - diagram 6-14
 - space significance 6-15
 - control statements 6-11
 - cos function
 - availability 6-1
 - loading procedure 6-15
 - define, keyword 6-16
 - define statement
 - built-in statement 6-22
 - description and use 6-23
 - demonstration run 6-2
 - described 6-1
 - division operator
 - left to right binding 6-19 , 6-5
 - scale 6-19 , 6-8
 - symbol (/) 6-5
 - equal sign (=)
 - assignment operator symbol 6-6
 - relational operator 6-12 , 6-21
 - equivalent constructions diagram 6-14
 - evaluation sequence 6-3
 - exclamation point (!)
 - relational operator 6-12 , 6-21
 - exit 6-2 , 6-4
 - exponential function
 - availability 6-1
 - loading procedure 6-15
 - exponentiation operator
 - right to left binding 6-19 , 6-5
 - scale 6-20 , 6-8

Index

- bc (*continued*)
 - exponentiation operator (*continued*)
 - symbol (^) 6-5
 - expression
 - enclosure 6-18
 - evaluation order 6-17
 - named expression 6-17
 - statement 6-22
 - for, keyword 6-16
 - for statement
 - break statement effect 6-23
 - built-in statement 6-22
 - description and use 6-11
 - format 6-23
 - range execution 6-12
 - relational operator 6-21
 - function
 - argument absence 6-11
 - array 6-11
 - call
 - defined 6-18
 - described 6-18
 - evaluation order 6-17
 - procedure 6-10
 - syntax 6-18
 - defined function 6-9
 - form 6-9
 - identifier 6-16
 - name 6-9
 - parameters 6-10
 - return statement 6-9
 - terminating, return statement 6-24
 - variable automatic 6-10
 - global storage class 6-21
 - greater-than sign (>), relational operator 6-12, 6-21
 - hexadecimal digit
 - ibase 6-7
 - obase 6-7
 - value 6-17
 - ibase
 - decimal input 6-7
 - defined 6-17
 - initial setting 6-6
 - keyword 6-16
 - named expression 6-17
 - setting 6-7
 - variable 6-9
 - identifier
 - array 6-22
 - auto statement effect 6-22
 - described 6-16
 - global 6-21
 - local 6-21
- bc (*continued*)
 - identifier (*continued*)
 - named expression 6-17
 - value 6-21
 - if, keyword 6-16
 - if statement
 - built-in statement 6-22
 - description and use 6-11
 - format 6-23
 - range execution 6-12
 - relational operator 6-21
 - INTERRUPT key 6-2
 - introduction 6-1
 - invoking 6-2
 - keywords designated 6-16
 - language features 6-14
 - length
 - built-in function 6-18
 - keyword 6-16
 - less-than sign (<), relational operator 6-12, 6-21
 - line continuation notation 6-7
 - local storage class 6-21
 - log function
 - availability 6-1
 - loading procedure 6-15
 - math function library 6-15
 - minus sign (-)
 - subtraction operator symbol 6-5
 - unary operator symbol 6-19, 6-5
 - modulo operator
 - left to right binding 6-19, 6-5
 - scale 6-19, 6-8
 - symbol (%) 6-5
 - multiplication operator
 - See also* specific operator
 - evaluation order 6-17
 - left to right binding 6-19, 6-5
 - scale 6-19, 6-8
 - symbol (*) 6-3, 6-5
 - named expression 6-17
 - negative number, unary minus sign (-) 6-5
 - obase
 - conversion speed 6-7
 - defined 6-17
 - described 6-7
 - hexadecimal notation 6-7
 - initial setting 6-7
 - keyword 6-16
 - named expression 6-17
 - variable 6-9
 - operator
 - See also* specific operator

bc (continued)

- operator (*continued*)
 - designated, use 6-5
- parentheses (())
 - expression enclosure 6-18
 - function identifier, argument enclosure 6-16
- percentage sign (%)
 - modulo operator symbol 6-5
- plus sign (+)
 - addition operator symbol 6-5
 - unary operator symbol 6-19
- program flow alteration 6-11
- quit command 6-4
- quit, keyword 6-16
- quit statement
 - bc exit 6-24
 - built-in statement 6-22
- quoted string statement 6-22
- register 6-3
- relational operator
 - designated 6-12 , 6-21
 - evaluation order 6-17
- return, keyword 6-16
- RETURN rt key 6-2
- return statement
 - built-in statement 6-22
 - described 6-24
 - form 6-9
- scale
 - addition operator 6-20 , 6-8
 - arctan function 6-15
 - Bessel function 6-15
 - built-in function 6-18
 - command 6-8
 - cos function 6-15
 - decimal digit value 6-9
 - defined 6-17
 - described 6-8
 - division operator 6-19 , 6-8
 - exponential function 6-15
 - exponentiation operator 6-20 , 6-8
 - initial setting 6-9
 - keyword 6-16
 - length function 6-18
 - length maximum 6-8
 - log function 6-15
 - modulo operator 6-19 , 6-8
 - multiplication operator 6-8
 - named expression 6-17
 - sin function 6-15
 - square root effect 6-18 , 6-8
 - subtraction operator 6-20 , 6-8
 - value printing procedure 6-9

bc (continued)

- scale (*continued*)
 - variable 6-9
- scale command 6-9
- semicolon (;), statement separation 6-22 , 6-4
- sin function
 - availability 6-1
 - loading procedure 6-15
- slash (/), division operator symbol 6-5
- space significance 6-15
- square root
 - built-in function 6-18
 - keyword 6-16
 - result as integer 6-6
 - scale procedure 6-8
 - sqrt keyword 6-16
- statement
 - See also* specific statement entry procedure 6-14
 - execution sequence 6-22
 - separation methods 6-22
 - types designated 6-22
- storage
 - classes 6-21
 - register 6-6
- subscript
 - array. *See* array
 - described 6-11
 - fractions discarded 6-11
 - truncation 6-16
 - value limits 6-11
- subtraction operator
 - left to right binding 6-5
 - scale 6-20 , 6-8
 - symbol (-) 6-5
- syntax 6-1
- token composition 6-16
- truncation 6-8
- unary operator
 - designated 6-19
 - evaluation order 6-17
 - left to right binding 6-19
 - symbol (-) 6-5
- value 6-17
- variable
 - automatic 6-10 , 6-21
 - name 6-9
 - subscripted 6-11
- while, keyword 6-16
- while statement
 - break statement effect 6-23
 - built-in statement 6-22
 - description cmd use 6-11

Index

- bc (*continued*)
 - while statement (*continued*)
 - executing 6-24
 - range execution 6-12
 - relational operator 6-21
 - bc command
 - bc, invoking 6-2
 - file, reading and executing 6-15
 - bc -l command, bc 6-15
 - BEGIN and END 11-11
 - Bessel function. *See* bc
 - bgnice option 9-17
 - /bin directory
 - command search 7-3
 - contents 7-45
 - name derivation 7-45
 - /usr/bin, files duplicated in 7-60
 - Binary logical
 - and operator 7-48
 - or operator 7-48
 - BINUNIQ shell procedure 7-60
 - BKSP
 - vi cursor movement 2-19
 - Bourne shell
 - TERM variable 2-57
 - terminal type 2-57
 - Braces ({ })
 - bc
 - compound statement enclosure 6-22
 - function body enclosure 6-9
 - command ({ }) 7-55
 - command grouping 7-33
 - pipeline use, enclosing a command list 7-27
 - variable
 - conditional substitution 7-52
 - enclosure 7-13
 - Brackets ([])
 - bc
 - array identifier 6-16
 - auto array 6-21
 - subscripted variable 6-11
 - directory name, not used in 7-4
 - ed metacharacter. *See* ed
 - metacharacter 7-4, 7-70
 - pattern matching 7-4
 - test command, used in lieu of 7-47
 - break command
 - for command control 7-32
 - loop control 7-32
 - shell built-in command 7-55
 - special shell command 7-40
 - while command control 7-32
 - Buffer
 - See* ed
 - See* vi
 - Built-in variables
 - awk 11-18, 11-56
 - Built-in Variables 11-18, 11-56, 11-8
- ## C
- c command. *See* ed
 - C language
 - bc
 - comment convention similarity 6-15
 - syntax agreement 6-1
 - shell language 7-2
 - c option
 - mail, carbon copy specification 4-5
 - shell, invoking 7-54
 - Calculation. *See* bc
 - Calculator functions. *See* bc
 - Calendar reminder service 4-24
 - Calling a remote terminal
 - See* ct command
 - Caret (^)
 - mail, first message, symbol 4-7
 - Caret (~)
 - bc, exponentiation operator symbol 6-5
 - ed use. *See* ed
 - case command
 - description and use 7-29
 - exit status 7-30
 - redirection 7-36
 - shell built-in command 7-55
 - Case delimiter symbol (;) 7-69
 - Case-part 7-69
 - cat command
 - ed use. *See* ed
 - cd command
 - argument 9-20
 - argument substitution 9-20
 - directory change
 - previous 9-20, 7-18
 - parentheses use 7-18
 - searches 7-58, 9-20
 - CDPATH variable 7-15, 9-20
 - Changing passwords 10-4
 - Character class. *See* ed
 - chmodsugid authorization 10-7
 - chown 10-7
 - chron option. *See* mail

- Close statement
 - awk 11-38
- Close Statement 11-38
- Colon (:):
 - command 7-40
 - mail
 - network mail 4-15
 - PATH variable use 7-14
 - shell built-in command 7-55
 - variable conditional substitution 7-53
 - vi use. *See* vi
- Colon command. *See* Colon (:), command
- COLUMNS variable 9-14, 9-19
- Combinations of Patterns 11-16
- Command
 - defined 7-26
 - delimiter. *See* ed
 - ed commands. *See* ed
 - enclosure in parentheses (()), effect 7-56
 - environment 7-20
 - execution
 - time 7-55, 7-2
 - exit status. *See* Exit status
 - grammar 7-68
 - grouping
 - exit status 7-35
 - parentheses (()) use 7-69
 - procedure 7-33
 - WRITEMAIL shell procedure 7-67
 - keyword parameter 7-20
 - line. *See* Command line
 - list. *See* Command list
 - mode. *See* vi
 - multiple commands 7-9
 - output substitution symbol 7-70
 - private command name 7-3
 - public command name 7-3
 - search
 - PATH variable 7-14
 - process 7-58
 - separation symbol (;) 7-69
 - shell, built-in commands 7-55
 - simple command
 - defined 7-2, 7-26
 - grammar 7-68
 - slash (/) beginning, effect 7-3
 - special shell commands
 - described 7-40
 - See* Shell
 - substitution
 - back quotation mark (`) 7-4
 - double quotation mark (") 7-5
- Command (*continued*)
 - substitution (*continued*)
 - procedure 7-9
 - redirection argument 7-6
 - vi commands. *See* vi
- Command line
 - awk 11-53
 - execution 7-24
 - options
 - See also* specific option
 - designated 7-54
 - pipeline, use in 7-27
 - rescan 7-24
 - scanning sequence 7-24
 - substitution 7-9
- Command Line 11-53
- Command line argument
 - awk 11-43
- Command lines
 - wide
 - Korn shell 9-19
- Command list
 - case command, execution 7-29
 - defined 7-26
 - for command, execution 7-31
 - grammar 7-68
- Command mode. *See* vi
- Command-line Arguments 11-43
- Commands
 - illegal 10-11
- Communication. *See* mail
- Comparison
 - awk 11-57
- Compose escape
 - See* mail
- continue command
 - for command control 7-32
 - shell built-in command 7-55
 - special shell command 7-40
 - until command control 7-33
 - while command control 7-32
- Control command
 - See also* specific control command
 - redirection 7-36
- Control flow statements
 - awk 11-53
- Control Flow Statements 11-27, 11-53
- Cooperation with shell
 - awk 11-45
- Cooperation with the Shell 11-45
- Copy
 - command 2-27
 - files
 - local site. *See* rcp

Index

Copy (*continued*)

- files (*continued*)

 - remote site. *See* uucp

- text 2-27

COPYPAIRS shell procedure 7-61

COPYTO shell procedure 7-61

crypt 10-16

csH command

- C-shell, invoking 8-2

C-shell

- > & symbol

 - redirecting 8-12

- |& symbol

 - redirecting 8-12

- alias command

 - listing 8-14

 - multiple command use 8-11

 - number limits 8-11

 - pipelines 8-11

 - quoting 8-11

 - removing 8-16

 - use 8-10, 8-14

- ampersand (&)

 - background job symbol 8-13

 - background job use 8-33

 - boolean AND operation (&&) 8-21

 - if statement, not used in 8-23

 - redirection symbol 8-12

- appending

 - noclobber variable effect 8-12

 - redirection symbol 8-12

- argument

 - expansion 8-31

 - group specification 8-33

- argv variable

 - filename expansion, preventing

 - 8-22

 - script contents 8-18

- arithmetic operations 8-21

- asterisk (*)

 - character matching 8-33

 - script notation 8-20

- background job

 - procedure 8-13

 - symbol (&) 8-13

 - terminating 8-13

- backslash (\)

 - filename, separating parts 8-33

 - if statement use 8-23

 - metacharacter

 - canceling 8-33

 - escape 8-11

 - separating parts of filenames 8-33

- boolean AND operation 8-21

C-shell (*continued*)

- boolean OR operation 8-21

- braces ({})

 - argument

 - expansion 8-31

 - grouping 8-33

- brackets ([])

 - character matching 8-33

- break command

 - foreach statement exit 8-25

 - loop break 8-22

 - while statement exit 8-25

- breaksw command

 - switch exit 8-25

- c command

 - reuse 8-7

- caret (^)

 - history substitution use 8-34

- character matching 8-33

- colon (:)

 - script modifier 8-24

 - substitution modifier use 8-34

- command

 - See also* specific command

 - break command 8-22

 - continue command

 - loop use 8-22

 - default argument 8-10

 - du command 8-13

 - execution status 8-21

 - expanding 8-32

 - file. *See* C-shell, script

 - foreach command

 - exit 8-25

 - script use 8-22, 8-29

 - history

 - See also* C-shell, history

 - use 8-14

 - history list 8-7

 - input supply 8-26

 - location

 - determining 8-14

 - recomputing 8-5

 - logout command 8-14, 8-2

 - multiple commands 8-13

 - prompt symbol (%) 8-3

 - quoting 8-29

 - read only option 8-28

 - reading from file 8-15

 - rehash command 8-5

 - repeating

 - mechanisms 8-9, 8-14

 - replacing 8-32

 - separating

- C-shell (*continued*)
 - command (*continued*)
 - symbol (;) 8-11 , 8-33
 - set command 8-4
 - similarity, foreach command 8-29
 - simplifying 8-10
 - source
 - command reading 8-15
 - substituting
 - string modification 8-24
 - symbol 8-34
 - termination testing 8-21
 - timing 8-15
 - transformation 8-10
 - unalias command 8-16
 - unset command 8-16
 - command prompt-symbol (%) 8-3
 - commands, multiple
 - alias use 8-11
 - single job 8-13
 - comment
 - metacharacter 8-34
 - script use 8-18
 - symbol 8-24
 - continue command
 - loop use 8-22
 - .cshrc file
 - alias placement 8-10
 - use 8-2
 - diagnostic output
 - directing 8-12
 - redirecting 8-12
 - directory
 - examination 8-5
 - listing 8-4
 - disk usage 8-13
 - dollar sign (\$)
 - last argument symbol 8-8
 - process number expansion 8-20
 - variable substitution
 - symbol 8-19
 - use 8-34
 - du command 8-13
 - :e modifier 8-24
 - echo option 8-28
 - else-if statement 8-24
 - environment
 - printing 8-15
 - setting 8-15
 - equal sign (=)
 - string comparison use (==), (=) 8-21
 - exclamation point (!)
 - history mechanism use 8-14 , 8-34 ,
- C-shell (*continued*)
 - exclamation point (!) (*continued*)
 - 8-8
 - noclobber, overriding 8-6
 - string comparison use (!=), (!~) 8-21
 - execute primitive 8-21
 - existence primitive 8-21
 - expansion
 - control 8-28
 - metacharacters designated 8-34
 - expression
 - enclosing 8-33
 - evaluation 8-21
 - primitives 8-21
 - extension, extracting 8-24
 - file
 - appending 8-12
 - command content 8-17
 - enquiries 8-21
 - overwriting
 - preventing 8-6
 - procedure 8-6
 - filename
 - expansion 8-31
 - expansion, preventing 8-22
 - home directory indicator 8-33
 - metacharacters designated 8-33
 - root extraction 8-24
 - scratch filename metacharacter 8-34
 - foreach command
 - exit 8-25
 - script use 8-22 , 8-29
 - goto
 - label
 - script cleanup 8-27
 - statement 8-25
 - greater-than sign (>)
 - redirection symbol 8-12 , 8-34
 - history
 - command
 - use 8-14 , 8-9
 - list
 - command substitution 8-14
 - contents display 8-14 , 8-7
 - mechanism
 - alias, use in 8-11
 - invoking 8-8
 - use 8-9
 - substitution symbol 8-34
 - variable 8-2
 - home variable 8-5
 - if statement 8-23

Index

C-shell (*continued*)

- ignoreeof variable 8-2 , 8-5
- input
 - execution procedure 8-19
 - metacharacters designated 8-34
 - variable substitution 8-19
- INTERRUPT key
 - background job, effect 8-13
- invoking 8-2
- kill command
 - background job termination 8-13
- less-than sign (<)
 - redirection symbol 8-34
 - script inline data supply (<<) 8-26
- logging out
 - logout command 8-14 , 8-2
 - procedure 8-3
 - shield 8-2
- .login file, use 8-2
- logout command
 - use 8-14 , 8-2
- .logout file, use 8-3
- loop
 - break 8-22
 - input prompt 8-29
 - variable use 8-29
- mail
 - invoking 8-2
 - variable
 - new mail notification 8-2 , 8-5
- metacharacter
 - cancelling 8-33
 - expansion metacharacter 8-34
 - filename metacharacter 8-33
 - input metacharacter 8-34
 - output metacharacter 8-34
 - quotation metacharacter 8-33
 - substitution metacharacter 8-34
 - syntactic metacharacter 8-33
- metasyntax
 - exclamation point (!) 8-6
- minus sign (-)
 - option prefix 8-34
- modifiers 8-24
- n key
 - out-of-range subscript errors,
 - absence 8-20
 - script notation 8-20
- n option 8-28
- new program, access 8-4
- noclobber variable
 - appending procedure 8-12
 - redirection symbols 8-12 , 8-5
- noglob variable

C-shell (*continued*)

- noglob variable (*continued*)
 - filename expansion, preventing 8-22
- number sign (#)
 - C-shell comment
 - symbol 8-18
 - use 8-24
 - C-shell comment symbol 8-28
 - C-shell comment use 8-34
 - scratch filename use 8-34
- onintr label
 - script cleanup 8-27
- option
 - metacharacter 8-34
- output
 - diagnostic 8-12
 - metacharacters designated 8-34
 - redirecting 8-12
- parentheses (())
 - enclosing an expression 8-33
- path variable 8-4
- pathname
 - component separation 8-33
- percentage sign (%)
 - command prompt symbol 8-3
- pipe symbol (|)
 - boolean OR operation (||) 8-21
 - command separator 8-33
 - if statement, not used in 8-23
 - redirection symbol 8-12
- pipeline
 - alias, use in 8-11
- primitives 8-21
- printenv
 - environment printing 8-15
- process number
 - expansion notation 8-20
 - listing 8-13
- prompt variable 8-14
- ps command
 - process number listing 8-13
- question mark (?)
 - character matching 8-33
 - loop input prompt 8-29
- QUIT signal
 - background job, effect on 8-13
- quotation mark
 - back (')
 - command use 8-29
 - substitutions 8-34
 - double (")
 - expansion control 8-28
 - double (_)

C-shell (*continued*)

- quotation mark (*continued*)
 - metacharacter escape 8-33
 - string quoting 8-29
- single (')
 - alias definition 8-11
 - metacharacter escape 8-33
 - quoted string, effect 8-28
 - script inline data quoting 8-26
- quotation metacharacters designated 8-33
- :r modifier 8-24
- read primitive 8-21
- redirecting
 - diagnostic output 8-12
 - output 8-12
 - symbols designated 8-34
- rhash command
 - command locations, recomputing 8-14 , 8-5
- repeat command 8-14
- root part of filename
 - separating from extensions 8-33
- script
 - clean up 8-27
 - colon (:) modifier 8-24
 - command input 8-26
 - comment required 8-28
 - described 8-17
 - example 8-22
 - execution 8-18
 - exit 8-27
 - inline data supply 8-26
 - interpretation 8-18
 - interruption catching 8-27
 - metanotation for inline data 8-26
 - modifiers 8-24
 - notations 8-20
 - range 8-20
 - variable substitution 8-19
- semicolon (;)
 - command separator 8-11 , 8-33
 - if statement, not used in 8-23
- set command
 - variable listing 8-4
 - variable value assignment 8-4
- setenv command
 - environment setting 8-15
- slash (/)
 - separating components of pathname 8-33
- source command
 - reading a command 8-15
- status variable 8-21

C-shell (*continued*)

- string
 - comparing 8-21
 - modifying 8-24
 - quoting 8-29
- substitution metacharacters
 - designated 8-34
- switch statement
 - exit 8-25
 - form 8-25
- syntactic metacharacters designated 8-33
- TERM variable 2-58
- terminal type, setting 2-58
- then statement 8-23
- tilde (~)
 - home directory indicator 8-33
- time
 - command timing 8-15
 - variable 8-2
- unalias command
 - alias, removing 8-16
- unset command 8-16
- unsettling procedure 8-5
- v command line option 8-28
- variable
 - See also* specific variable
 - component access
 - notations 8-19
 - definition
 - removing 8-16
 - environment variable setting 8-15
 - expansion 8-19 , 8-30
 - listing 8-4
 - loop use 8-29
 - setting procedure 8-5
 - substitution
 - metacharacter 8-34 , 8-19
 - use 8-4
 - value assignment
 - check 8-19 , 8-4
- verbose option 8-28
- while statement
 - exit 8-25
 - form 8-25
 - write primitive 8-21
- x command line option 8-28
- C-shell with UUCP commands 5-9
- .cshrc file
 - C-shell use 8-2
- ct command
 - h option 5-17
 - how it works 5-15
 - s option 5-16

Index

ct command (*continued*)
 sample command 5-16
 syntax of 5-15
 using 5-15
 when to use 5-15

Ctrl-d
 bc exit 6-2, 6-4

Ctrl-D
 mail
 message sending 4-10
 shell exit 4-15

Ctrl-d
 shell exit 7-33
 vi, scroll 2-23

Ctrl-f
 vi, scroll 2-23

Ctrl-g
 vi, file status information 2-11

Ctrl-U
 mail, line kill 4-9

Ctrl-u
 vi, scroll 2-22

Ctrl-v 2-62

Ctrl-z
 Korn shell
 job control 9-10

cu command
 calling
 UNIX sites 5-17
 command line 5-17
 dialing phone numbers with 5-17
 error checking 5-20
 interactive sessions with 5-17
 limitations on 5-17
 logging in with 5-19
 put command 5-19
 sample command 5-18
 serial lines with 5-18
 syntax of 5-17
 system names with 5-18
 take command 5-19
 terminating a remote session 5-18
 transfer files 5-19
 using 5-17, 5-18

current file (%) 2-62

Current line
 See vi

Cursor movement
 vi. *See* vi

Cut and paste procedure. *See* ed

D

d command
 cd use. *See* ed
 mail
 message deleting 4-8

D command. *See* vi

d0 command. *See* vi

dd command. *See* vi

Decrypting a File 10-18

Delete
 commands 2-66
 vi procedure. *See* vi, deleting text

Delete buffer. *See* vi

Delimiter. *See* ed

Diagnostic output. *See* Output

Dial-up line. *See* Background process

Digit grammar 7-69

Directory
 change. *See* cd command

C-shell
 listing 8-4
 use. *See* C-shell

name, metacharacters in 7-4

return to previous 9-20

search
 optimum order 7-58
 PATH variable 7-58
 sequence change 7-3
 size effect 7-59
 time consumed in 7-58
 size consideration 7-59

DISTINCT1 shell procedure 7-62

Division. *See* bc

Dollar sign (\$) *See* ed
 ed use. *See* ed
 mail, final message, symbol 4-7
 positional parameter prefix 7-11, 7-12
 PS1 variable default value 7-15
 variable prefix 7-12
 vi use. *See* vi

DOS
 compile UNIX programs for DOS 14-3
 file
 access to 14-1
 filesystems
 access to 14-1
 utilities
 access to 14-1
 using 14-2

Dot (.)
 command

Dot (.) (*continued*)
 command (*continued*)
 description and use 7-36
 shell built-in command 7-55
 shell procedure alternate 7-45
 special shell command 7-40
 ed use. *See* ed
 mail, current message, symbol 4-7
 vi use. *See* vi
 Dot command. *See* Dot (.), command
 dp command. *See* mail
 DRAFT shell procedure 7-63
 dw command. *See* vi

E

e command
 ed use. *See* ed
 mail 4-4
 -e option, shell procedure 7-46
 echo command
 description and use 7-49
 -n option effect 7-49
 shell built-in command 7-55
 syntax 7-49
 ed
 a command
 appending 3-5, 3-58
 backslash (\) characteristics 3-38
 dot (.) setting 3-50, 3-58
 global combination 3-29
 terminating input 3-36, 3-5
 address arithmetic 3-11
 ampersand (&)
 literal 3-47
 metacharacter 3-45
 substitution 3-45
 appending
 a command 3-5
 asterisk (*), metacharacter 3-33, 3-41
 at sign (@), script 3-57
 backslash (\)
 a command 3-38
 c command 3-38
 g command 3-28
 i command 3-38
 line folding 3-30
 literal 3-37
 metacharacter 3-33, 3-36
 metacharacter escape 3-36, 3-37, 3-47
 multiline construction 3-29
 ed (*continued*)
 backslash (\) (*continued*)
 number string 3-30
 v command 3-28
 backspace printing 3-29
 brackets ([])
 character class 3-45
 metacharacter 3-33, 3-44
 buffer
 described 3-5
 writing to file 3-6
 c command
 backslash (\) characteristics 3-38
 dot (.) setting 3-24, 3-50, 3-58
 global combination 3-29
 line change 3-23, 3-58
 terminating input 3-23
 caret (^)
 character class 3-45
 line beginning notation 3-41
 metacharacter 3-33, 3-41
 cat command 3-8
 change command
 c command 3-23
 character
 class 3-45
 deleting 3-44
 command
See also specific command
 combinations 3-28
 delimiter character 3-38
 described 3-5
 editing command 3-56
 form 3-58
 INTERRUPT key effect 3-54
 listing 3-58
 multicommand line restrictions 3-17
 summary 3-58
 current line 3-13
 cut and paste
 move command 3-25
 procedures 3-55
 d command
 deleting 3-15, 3-58
 dot (.) setting 3-50, 3-58
 DEL key
 print stopping 3-11
 deleting
 d command 3-15
 delimiter
 character choice 3-38
 described 3-1
 dollar sign (\$)

Index

ed (*continued*)

- dollar sign (\$) (*continued*)
 - last line notation 3-10 , 3-15 , 3-40
 - line end notation 3-39 , 3-40
 - metacharacter 3-33 , 3-39
 - multiple functions 3-40
- dot (.)
 - current line notation 3-11
 - described 3-13
 - position in file 3-50
 - search setting 3-20 , 3-60
 - substitution, setting 3-17
 - symbol (.) 3-13 , 3-36
 - value determination 3-14 , 3-59
- duplication
 - t command 3-31
- e command 3-58 , 3-8
- editing
 - e command 3-8
- entry 3-4
- equals sign (=)
 - dot value printing (=) 3-14 , 3-59
 - last line value printing 3-59
- escape command (!) 3-32 , 3-59
- exclamation point (!)
 - escape command 3-32
- exiting a file
 - q command 3-4
- f command 3-58 , 3-8
- file
 - insert into another file 3-55
 - writing out 3-55
- filename
 - change 3-8
 - recovery 3-8
 - remembered filename printing 3-58
 - remembered filename, printing 3-8
- folding 3-30
- g command
 - a command combination 3-29
 - backslash (\) use 3-28
 - c command combination 3-29
 - command combinations 3-27 , 3-28
 - dot (.) setting 3-28
 - i command combination 3-29
 - line number specifications 3-28
 - multiline construction 3-29
 - s command combination 3-27 , 3-59
 - search, command execution 3-26 , 3-58
 - substitution 3-18 , 3-33
 - trailing g 3-33
- global command

ed (*continued*)

- global command (*continued*)
 - g command 3-26
 - v command 3-26
- greater-than sign (>), tab notation 3-29
- grep command 3-33
- hyphen (-), character class 3-45
- i command
 - backslash (\) characteristics 3-38
 - dot (.) setting 3-24 , 3-50 , 3-58
 - global combination 3-29
 - inserting 3-23 , 3-58
 - terminating input 3-36
- in-line input scripts 7-63
- input
 - terminating 3-23 , 3-36 , 3-5
- inserting
 - i command 3-23
- INTERRUPT key
 - command execution effect 3-54
- dot (.) setting 3-54
- introduction 3-1
- invoking 3-4
- j command
 - line joining 3-48
- k command, line marking 3-30
- l command
 - folding 3-30
 - line listing 3-29 , 3-58
 - nondisplay character printing 3-29
 - number string 3-30
 - s command combination 3-34
- less-than sign (<)
 - backspace notation 3-29
- line
 - beginning
 - character deleting 3-44
 - notation 3-41
 - break 3-47
 - end
 - notation 3-39
 - folding 3-30
 - joining 3-48
 - marking 3-30
 - moving 3-30
 - new 3-47
 - number
 - 0 as line number 3-54
 - combinations 3-11
 - summary 3-58 , 3-11
 - rearrangement 3-48
 - splitting 3-47
 - writing out 3-56

- cd (*continued*)
 - list
 - l command 3-29
 - m command
 - dot (.) setting 3-26 , 3-59
 - line moving 3-25 , 3-59
 - warning 3-26
 - mail system. *See* mail
 - marking
 - k command 3-30
 - metacharacter
 - ampersand (&) 3-45
 - asterisk (*) 3-33 , 3-41
 - backslash (\) 3-33 , 3-36
 - brackets ([]) 3-33 , 3-44
 - caret (^) 3-33 , 3-41
 - character class 3-45
 - combinations 3-41
 - dollar sign (\$) 3-33 , 3-39
 - escape 3-38 , 3-47
 - period (.) 3-33 , 3-34
 - search 3-45
 - slash (/) 3-33
 - star (*) 3-33 , 3-41
 - minus sign (-), address arithmetic 3-11
 - move
 - line marking 3-30
 - m command 3-25
 - multicommand line restrictions 3-17
 - new line
 - substitution 3-47
 - nondisplay character printing 3-29
 - p command
 - dot (.) setting 3-54
 - multicommand line 3-17
 - printing 3-10 , 3-59
 - s command combination 3-34
 - pattern search. *See* ed, search
 - period (.)
 - a command, terminating input 3-36 , 3-5
 - c command
 - terminating input 3-23
 - character substitution 3-34
 - dot symbol. *See* Dot (.)
 - i command, terminating input 3-36
 - literal 3-36
 - metacharacter 3-33 , 3-34
 - s command, effect 3-34
 - script problems 3-57
 - search problems 3-33
 - troff command prefix 3-27
 - plus sign (+), address arithmetic 3-11
- ed (*continued*)
 - print
 - command 3-10
 - line folding 3-30
 - (Return
 - key effect 3-14
 - stopping 3-11
 - q command
 - abort 3-59
 - quit session 3-59 , 3-7
 - w command combination 3-59
 - question mark (?)
 - exit warning 3-4
 - search error message (?) 3-20
 - search repetition (??) 3-22
 - search, reverse direction (? ?) 3-20 , 3-60
 - write warning 3-7
 - quit
 - q command 3-7
 - quotation mark, single (')
 - line marking 3-30
 - r command
 - dot (.) setting 3-51 , 3-59
 - file inserting 3-55
 - positioning without address 3-55
 - read file 3-59 , 3-9
 - reading
 - r command 3-9
 - regular expression
 - described 3-33
 - metacharacter list 3-33
 - RETURN key, printing 3-59
 - s command
 - ampersand (&) 3-45
 - character match 3-34
 - description and use 3-16 , 3-59
 - dot (.) setting 3-17 , 3-50 , 3-59
 - g command combination 3-18 , 3-27 , 3-59
 - l command combination 3-34
 - line number 3-34
 - new line 3-47
 - p command combination 3-34
 - removing text 3-18
 - search combination 3-21
 - trailing g 3-33
 - undoing 3-30
 - v command combination 3-27
 - script 3-57
 - search
 - dot (.) setting 3-60
 - error message (?) 3-20
 - forward search (/ /) 3-19 , 3-60

Index

ed (continued)

search (continued)

global search

g command 3-27

v command 3-27

metacharacter problems 3-33

next occurrence description 3-20

procedure 3-19

repetition (//), (??) 3-22

reverse direction (? ?) 3-20

separator 3-52

substitution combination 3-21

sed command 3-33

semicolon (;)

dot (.) setting 3-53

search separator 3-52

shell

escape 3-32

slash (/)

delimiter 3-38

literal 3-38

metacharacter 3-33

search forward (//) 3-19, 3-60

search repetition (//) 3-22

special character 3-33

spelling correction

s command 3-16

star (*), metacharacter 3-33, 3-41

substituting

s command 3-16

t command

dot (.) setting 3-59

transfer line 3-31, 3-59

tab printing 3-29

tbl command 3-56

terminating

q command 3-7

text

removing, s command 3-18

saving 3-7

transfer 3-31

troff command printing 3-27

typing-error corrections

s command 3-16

u command

undo 3-30, 3-59

undo

u command 3-30

v command

a command combination 3-29

backslash (\) use 3-28

c command combination 3-29

command combinations 3-27, 3-28, 3-29

ed (continued)

v command (continued)

dot (.) setting 3-28

global search, substitute 3-26, 3-59

i command combination 3-29

line number specifications 3-28

s command combination 3-27

w command

advantages of frequent use 3-52

description and use 3-6

dot (.) setting 3-51, 3-59

e command combination 3-58

file write out 3-55

line write out 3-56

write out 3-55, 3-59, 3-6, 3-7

write out

w command 3-7

warning 3-7

ed scripts 12-1

ed -x 10-16

EDFIND shell procedure 7-63

edit -x 10-17

Editor

Korn shell command-line 9-3

See ed

See vi, described

EDITOR variable 9-14, 9-3

EDLAST shell procedure 7-64

elif clause, if command 7-28

else clause, if command 7-28

Else-part grammar 7-69

emacs option 9-17

Empty grammar 7-69

Encrypting a File 10-17

Encryption

data encryption commands 10-16

definition 10-16

ENV variable 9-14

Equal sign (=)

bc

assignment operator symbol 6-6

relational operator 6-12, 6-21

ed use. See ed

variable

conditional substitution 7-52

string value assignment 7-12

error messages

awk 11-10

Error Messages 11-10

Error output

redirecting 7-51

ESC key 2-62

escape command (!) 3-32

etc/default/micnet 5-5

- /etc/default/msdos file
 - contents 14-3
- eval command
 - command line rescan 7-24
 - shell built-in command 7-55
- ex and ed, similarity 3-1
- ex -x 10-17
- Example applications
 - awk 11-48
- Example Applications 11-48
- Exclamation point (!)
 - bc, relational operator 6-12 , 6-21
 - C-shell use. *See* C-shell
 - ed use. *See* ed
 - mail
 - network mail 4-16
 - shell command, executing 4-15
 - unary negation operator 7-48
 - vi use. *See* vi
- exec command 7-40 , 7-55
- execsuid 10-7
- Execute
 - commands
 - over Micnet. *See* remote
 - remote machines. *See* uux
 - command
- EXINIT environment variable 2-57
- Exit
 - code 7-16
 - command. *See* exit command
 - status
 - \$? variable 7-16
 - case command 7-30
 - cd arg command 7-40
 - colon command (:) 7-40
 - command grouping 7-35
 - false command 7-50
 - if command 7-28
 - read command 7-41
 - true command 7-50
 - until command 7-30
 - wait command 7-43
 - while command 7-30
- exit command
 - shell built-in command 7-55
 - shell exit 7-33
 - special shell command 7-40
- export command
 - shell built-in command 7-55
- variable
 - example 7-16
 - listing 7-21
 - setting 7-20
- expr command 7-49

F

- f command
 - ed use. *See* ed
 - mail 4-14
- F command, mail 4-14
- f option
 - mail, folder specification 4-13
- false command 7-50
- fc command 9-8
- fi command
 - if command end 7-28
- Field
 - awk 11-4
- Field variable
 - awk 11-25
- Field Variables 11-25
- Fields 11-4
- File
 - creating
 - MKFILES shell procedure 7-65
 - with vi 2-2
 - descriptor
 - redirection 7-51 , 7-7
 - use 7-6
 - grammar 7-69
 - pattern search
 - grep command 3-59
 - See* ed, search
 - pipe interchange 7-62
 - shell procedure, creating 7-44
 - textual contents, determining 7-67
 - variable file, creating 7-36
- file command
 - mail 4-13
- Filename
 - argument 7-3
 - ed use. *See* ed
- Files
 - security 10-13
- Files and Pipes
 - awk 11-39
- Filter
 - described 7-8
 - order consideration 7-57
- Flag. *See* Option
- folder command
 - mail 4-13
- Folder directory. *See* mail
- folder= option
- folder. *See* mail
- folder 4-13
- for command

Index

for command (*continued*)

- break command effect 7-32
- continue command effect 7-32
- description and use 7-31
- redirection 7-36
- shell built-in command 7-55

for loop, argument processing 7-23

fork command 7-56

Formatted printing

- awk 11-6

Formatted Printing 11-6

Form-letter generation

- awk 11-52

FSPLIT shell procedure 7-64

Function

- defined 7-35

Functions

- awk 11-54, 11-9, 11-54, 11-9

G

G command

- vi use. *See* vi
2-5 "

g command. *See* ed

Generating reports

- awk 11-48

Generating Reports 11-48

getline function

- awk 11-40

Global

substitution

- ed use. *See* ed
- See* vi, search and replace
vi 2-41

- variable check 7-46

global substitution 12-1

goto command 2-5

Greater-than sign (>)

- bc, relational operator 6-12, 6-21
- PS2 variable default value 7-15
- redirection symbol 7-69

grep command

- ed use. *See* ed

gsub 11-22

H

h command

- mail 4-6
- vi use. *See* vi

hash command

- described 7-41
- special shell command 7-41

HISTFILE variable 9-14, 9-17

history command

- C-shell 8-9

History command 9-8

history file

- Korn shell 9-17

HISTSIZ variable 9-14, 9-17

HOME variable

- conditional substitution 7-53
- described 7-13, 9-14

I

i command. *See* ed

-i option

- shell, invoking 7-54

if command

- COPYTO shell procedure 7-62
- description and use 7-28
- exit status 7-28
- fi command required 7-28
- multiple testing procedure 7-28
- nesting 7-28
- redirection 7-36
- shell built-in command 7-55
- test command 7-47

IFS variable 7-13

ignorecase option 2-41

ignoreeof option 9-17

Index

- length 11-23

Indirect file transfers over phone lines
5-10

Initialization

- awk 11-57

Initialization, comparison, and type
coercion

- awk 11-57

Initialization, Comparison, and Type
Coercion 11-57

In-line input document. *See* Input

Input

- awk 11-39

Input (*continued*)
 ed use. *See* ed
 grammar 7-68
 in-line input
 document 7-50
 EDFIND shell procedure 7-63
 standard
 input file 7-6 , 11-39
 input separators
 awk 11-39
 Input Separators 11-39
 Input-output
 awk 11-54
 Insert
 See also Append
 ed use. *See* ed
 vi procedure 2-25
 Insert mode. *See* vi
 Internal field separator
 shell scanning sequence 7-24
 specified by IFS variable 7-13
 Interrupt
 handling methods 7-36
 key. *See* INTERRUPT key
 INTERRUPT key
 background process immunity 7-26
 bc 6-2
 ed use. *See* ed
 mail
 askcc switch 4-23
 cancel 4-10
 Introduction
 awk 11-1
 Invocation flag. *See* Option
 Item grammar 7-68

J

j command
 ed use. *See* ed
 j command, cursor movement
 vi use. *See* vi
 J command, joining lines
 vi use. *See* vi
 Job control
 Korn shell 9-10

K

k command
 ed use. *See* ed
 vi use. *See* vi
 -k option, shell procedure 7-46
 kernel authorizations
 nopromin 10-7
 Keyword parameter
 described 7-20
 -k option effect 7-46
 kill command
 C-shell use. *See* C-shell
 Korn shell
 aliases 9-15
 cd command enhancements 9-20
 command-line
 editors 9-3
 description 9-1
 editors
 command-line 9-3
 emacs editor 9-3
 environment file 9-3
 environment variables 9-14 , 9-3
 history file 9-14 , 9-17 , 9-2 , 9-8
 invoking 9-2
 job control
 commands 9-11 , 9-10
 .kshrc file 9-15 , 9-3
 login shell 9-2 , 9-4
 options 9-16
 .profile file 9-14
 r command 9-8
 reexecuting commands 9-8
 .sh_history file 9-2 , 9-8
 vi editor
 control mode 9-5
 input mode 9-4 , 9-3 , 9-4
 wide command lines 9-19
 ksh command. *See* Korn shell
 .kshrc file 9-15 , 9-3

L

l command
 ed use. *See* ed
 mail 4-13
 vi use. *See* vi
 Less-than sign (<)
 bc, relational operator 6-12 , 6-21
 redirection symbol 7-69

Index

Limits

- awk 11-56

Line

- beginning. *See* ed
- writing out. *See* ed

line command

- shell variable value assignment 7-10

- linenumber option. *See* vi

- Line-oriented commands 2-11

- list option. *See* vi

- LISTFIELDS shell procedure 7-65

- Locked account 10-3

- Locked terminal 10-3

- Logging in 10-13

- Logging out

- shell termination 7-33

- Login directory

- defined 7-13

- .login file

- C-shell use 8-2

- Login Security 10-3

- login spoofing program 10-1

- logout command

- C-shell use 8-2

- .logout file

- C-shell use 8-3

- .logout file 9-15

Looping

- break command 7-32

- continue command 7-32

- control 7-32

- expr command 7-50

- false command 7-50

- for command 7-31

- iteration counting procedure 7-50

- time consumed in 7-55

- true command 7-50

- unconditional loop implementation
7-50

- until command 7-30

- while command 7-30

- while loop 7-61

- lp command

- mail

- m option 4-24

- lpr command

- mail

- message printing 4-13

- ls command

- echo *, used in lieu of 7-49

M

- m command

- ed use. *See* ed

- M flag, mail 4-6

- m option, mail 4-24

- magic option. *See* vi

- mail

- ~? help escape 4-18

- ~! shell escape 4-15

- a command. *See* mail, alias

- accumulation of 4-25

- alias

- a command 4-21, 4-22

- display 4-21, 4-22

- personal 4-21, 4-22

- R command 4-22

- system-wide 4-21

- askcc option 4-23

- asterisk (*)

- message saved, header notation

- 4-12, 4-6

- ~b escape 4-19

- BACKSPACE key 4-9

- Bcc field 4-19

- blind carbon copy field

- described 4-3

- editing 4-19

- box. *See* Mailbox

- ~c escape 4-19

- c option 4-5

- carbon copy

- c option 4-5

- carbon copy field

- blind field 4-3

- described 4-3

- editing 4-19

- escape

- ~c escape 4-19

- option

- askcc option 4-23, 4-11

- caret (^), first message, symbol 4-7

- cc field 4-19

- character matching. *See* mail

- metacharacters

- chron option 4-23

- colon (:)

- network mail 4-15

- command

- See also* the specific command

- invoking 4-7

- mode

- description and use 4-3

- mail (*continued*)
 - command (*continued*)
 - syntax 4-7
 - command line, options 4-5
 - compose
 - escape
 - See also* the specific escape
 - edit mode 4-4
 - heading escape 4-18
 - listing 4-18
 - tilde (~) component 4-18 , 4-4
 - compose mode
 - description and use 4-3
 - edit mode, entering 4-4
 - entering from
 - shell 4-9
 - concepts 4-2
 - C-shell
 - new mail notification 8-2
 - Ctrl-D
 - message sending 4-10
 - Ctrl-U, line kill 4-9
 - current message 4-6
 - d command
 - message deleting 4-8 , 4-14
 - ~d escape 4-19 , 4-20
 - ~dead escape 4-19
 - dead.letter file
 - aborted message 4-10
 - escape 4-20
 - distribution list, creating 4-21 , 4-22
 - dollar sign (\$)
 - final message, symbol 4-7
 - dot (.), current message symbol 4-7
 - dp command 4-14
 - e command 4-4
 - ~e escape 4-4
 - editor escapes 4-4
 - escape
 - editing 4-4
 - headers 4-18 , 4-19
 - help 4-18
 - printing 4-9
 - tilde escapes 4-18 , 4-4
 - exclamation point (!)
 - network mail 4-16
 - shell command, executing 4-15
 - exit
 - q command 4-12 , 4-17
 - x command 4-17
 - F command 4-14
 - f option 4-13
 - file command 4-13
 - folder 4-13
- mail (*continued*)
 - folder command 4-13
 - folder= option 4-25
 - forwarding
 - messages not deleted 4-17
 - procedure 4-14
 - h command 4-6
 - ~h escape 4-19
 - header
 - characteristics 4-6
 - command 4-6
 - compose escape 4-18
 - defined 4-6
 - display 4-10 , 4-5
 - windows 4-6
 - ~headers escape 4-19
 - heading
 - composition 4-3
 - help
 - command (?) 4-11
 - escape (~?) 4-18
 - hold command
 - P flag 4-6
 - hold option
 - description and use 4-23
 - effect 4-17
 - INTERRUPT key
 - cancel 4-10
 - recipient list 4-23
 - l command 4-13
 - line kill 4-9
 - lp command
 - m option 4-24
 - lpr command
 - message printing 4-13
 - ~M escape 4-20
 - M flag, message saving 4-6
 - m option 4-24
 - mail command
 - command mode entry 4-3
 - help 4-11
 - message reading 4-10
 - message sending 4-3
 - mail escapes 4-20
 - .mailrc file
 - alias command 4-21
 - distribution list, creating 4-22
 - example 4-21
 - set command 4-21
 - unset command 4-21
 - message
 - body 4-3
 - cancel 4-10
 - composition 4-3

Index

mail (*continued*)

- message (*continued*)
 - delete, undoing 4-14
 - deleting 4-14 , 4-8
 - described 4-3
 - editing 4-4 , 4-9
 - file, including a 4-19
 - header 4-6
 - inserting into new message 4-20
 - list. *See* mail, message list
 - printing 4-14 , 4-7
 - range described 4-7
 - reading 4-10
 - saving 4-12 , 4-8
 - sending 4-10
 - specification 4-7
- message list
 - composition 4-7
- message types 4-7
- metacharacters 4-7
- Micnet 4-15
- network mail 4-15
- number command 4-7
- options
 - command-line options 4-5
 - setting 4-21
- organizing 4-25
- p command
 - message printing 4-7
- ~p escape 4-9
- P flag, message preserving 4-6
- period (.), dot use 4-7
- preserve command
 - P flag 4-6
- printing
 - lineprinter, lpr command 4-13
 - ~p escape 4-9
- prompt= option 4-11
- q command
 - exit 4-12 , 4-17
- question mark (?)
 - compose escape help 4-18
- R command
 - alias effect 4-22
 - message reply 4-13
- ~r escape 4-19
- read escape
 - ~d escape 4-19
 - ~r escape 4-19
 - ~read escape 4-19
- recipient list, adding a name 4-18
- reminder service 4-24
- reply command 4-13
- s command

mail (*continued*)

- s command (*continued*)
 - flag 4-6
 - message saving 4-12 , 4-8
 - saving 4-17 , 4-6
- ~s escape 4-18
- s option 4-5
- saving
 - asterisk (*) notation 4-12
 - automatic 4-14
 - flag 4-6
 - M flag 4-6
- screen= option 4-6
- search string 4-7
- sending
 - cancellation impossible 4-10
 - network mail 4-15
 - procedure 4-9
 - remote sites
 - UUCP 4-16 , 4-15
- session abort 4-17
- set command
 - description and use 4-21
- set options defined 4-21
- sh command 4-15
- shell
 - commands 4-15
 - escapes 4-15
- special characters. *See* mail metacharacters
- star (*), asterisk use 4-8
- startup file 4-21
- status indicator 4-6
- string option
 - setting 4-21
- subject
 - asksub option 4-9
 - escape 4-18
 - field 4-3
 - s option 4-5
- subject field 4-11
- switch option, setting 4-21
- ~t escape 4-18
- tilde
 - compose escapes 4-18 , 4-4
- to field
 - mandatory 4-3
- u command 4-14
- undeleting. *See* mail, u command
- undeliverable message 4-10
- unset command
 - description and use 4-21
- UUCP 4-15
- v command 4-4

- mail (*continued*)
 - ~v escape 4-4
 - MAIL
 - variable. *See* MAIL variable
 - mail
 - vi
 - entering from compose mode 4-4
 - w command
 - flag 4-6
 - writing 4-17, 4-6
 - writing
 - flag 4-6
 - x command
 - exit 4-17
 - session abort 4-17
 - mail command
 - advantages of using 5-5
 - disadvantages of using 5-5
 - transferring files with 5-5, 4-3, 5-2, 5-5
 - MAIL variable 7-14
 - Mailbox
 - cleaning out 4-25
 - system mailbox 4-2
 - user mailbox
 - filename 4-2
 - message saving notation 4-6
 - MAILCHECK variable 7-14, 9-14
 - MAILPATH variable 7-14
 - makekey 10-16
 - map command 2-62
 - Marking. *See* ed
 - Match 11-23
 - mbox file. *See* Mailbox
 - mem authorization 10-8
 - mesg option. *See* vi
 - Message types. *See* mail, message list
 - Metacharacter
 - asterisk (*) 7-70
 - brackets ([]) 7-70
 - directory name, not used in 7-4
 - escape 7-4
 - list 7-69
 - mail 4-7
 - question mark (?) 7-70
 - redirection restriction 7-7
 - Micnet network 5-2
 - Minus sign *-)
 - bc
 - subtraction operator symbol 6-5
 - Minus sign (-)
 - bc
 - unary operator symbol 6-19, 6-5
 - redirection effect 7-50
 - Minus sign (-) (*continued*)
 - subtraction operator symbol 6-5
 - variable conditional substitution 7-52
 - Mistakes, correcting 2-25
 - MKFILES shell procedure 7-65
 - monitor option 9-17
 - Multi-line records
 - awk 11-40
 - Multi-Line Records 11-40
 - Multiple way branch 7-29
 - Multiplication. *See* bc
- ## N
- n command. *See* vi
 - n option
 - echo command 7-49
 - shell procedure 7-46
 - Name grammar 7-69
 - newgrp command
 - described 7-41
 - special shell command 7-41
 - Newline substitution. *See* ed
 - next command. *See* vi 2-50
 - nohup command 7-26
 - noprmain authorization 10-7
 - Notational conventions 1-4
 - nu command. *See* vi 2-27
 - Null command. *See* Colon (:), command
 - NULL shell procedure 7-66
 - Number or String? 11-26
 - Number sign (#), comment symbol 7-69
- ## O
- o operator 7-48
 - Operator. *See* bc
 - Operators
 - awk 11-55
 - Operators)Increasing Precedence)
 - 11-55
 - Option
 - See also* specific option
 - DRAFT shell procedure 7-63
 - invocation flags 7-54
 - Korn shell. *See* Korn shell
 - tracing, \$- variable 7-17
 - vi options. *See* vi
 - Or-if operator (||)

Index

Or-if operator (||) (*continued*)
command list 7-26
described 7-27
designated 7-69

Output

append symbol (>>) 7-6
append symbol (>>) 7-69
awk 11-34
creation symbol (>) 7-69
diagnostic output file 7-6
error redirection 7-51
grammar 7-68
standard
error file 7-6
output file 7-6, 11-34

output into files

awk 11-37

Output into Files 11-37

Output into Pipes 11-37

Output separators

awk 11-34

Output Separators 11-34

P

p command

ed use. *See* ed

mail

message printing 4-7

P flag, mail 4-6

Parentheses (())

bc

expression enclosure 6-18

function identifier, argument

enclosure 6-16

command grouping 7-33, 7-56, 7-69

pipeline use, command list 7-27

test command operator 7-48

Password security 10-12

PATH variable

conditional substitution 7-53

C-shell use. *See* C-shell

described 7-14

directory search

effect 7-58

sequence change 7-3, 9-14

Pattern

grammar 7-69

metacharacter 7-70

Pattern matching facility

case command 7-29

expr command argument effect 7-49

Pattern matching facility (*continued*)

limitations 7-4

metacharacter. *See* Metacharacter

redirection restriction 7-6

shell function 7-3

variable assignment, not applicable
7-12

pattern ranges

awk 11-17

Pattern Ranges 11-17

Patterns

awk 11-11, 11-16, 11-53

patterns

awk 11-6

Patterns

BEGIN and END 11-11, 11-53

percent sign, current file 2-62

Percentage sign (%)

bc modulo operator symbol 6-5

Period (.)

See also Dot (.)

ed use. *See* ed

pattern matching facility, restrictions
7-4

vi use. *See* vi

PHONE shell procedure 7-66

PID

\$\$ variable 7-16

#! variable 7-17

Pipe

file interchange 7-62

symbol (|) 7-69

Pipeline

command list 7-27

C-shell use. *See* C-shell

defined 7-26

described 7-7

DISTINCT1 shell procedure 7-62

filter 7-8

grammar 7-68

notation 7-7

procedure 7-7

pipes

awk 11-37

output into 11-37

Plus sign (+)

bc

addition operator symbol 6-5

unary operator symbol 6-19

variable, conditional substitution 7-53

Positional parameter

assignment statement positioning

7-12

described 7-11

Positional parameter (*continued*)
 direct access 7-23
 null value assignment 7-52
 number yield, \$# variable 7-16
 parameter substitution 7-12
 positioning 7-12
 prefix (\$) 7-12
 setting 7-11

Print
 command. *See* p command
 ed use. *See* ed

Print statement
 awk 11-34
 printerstat authorization 10-8
 printf
 awk 11-35, 11-36, 11-6

Printing
 awk 11-4
 formatted 11-6, 11-4
 printqueue authorization 10-8

Process
 defined 7-2
 number. *See* PID
 profile file
 description and use 7-19
 Korn shell 9-14
 PATH variable setting 7-15
 variable export 7-16

Program structure
 awk 11-2

Program Structure 11-2

Prompt. *See* mail
 prompt= option
 protected subsystem 10-2

ps command
 C-shell use. *See* C-shell

PS1 variable 7-15, 9-14
 PS2 variable 7-15

Q

q command
 bc 6-2
 ed exit. *See* ed
 mail
 exit 4-12, 4-17

q!. *See* vi

qucrspace authorization 10-8

Question mark (?)
 directory name, not used in 7-4
 ed use. *See* ed
 mail

Question mark (?) (*continued*)
 mail (*continued*)
 compose escapes, listing 4-18
 help command 4-11
 metacharacter 7-4, 7-70
 pattern matching
See Question mark (?),
 metacharacter
 variable conditional substitution 7-53

quit command
 bc exit 6-2, 6-4
 q command 6-2

QUIT key, background process
 immunity 7-26

Quotation mark
 back (`)
 command substitution 7-10, 7-4
 quoting 7-70
 double (")
 metacharacter escape 7-4
 double (_)
 quoting 7-70
 test command 7-47
 variable 7-12
 single (')
 C-shell use. *See* C-shell
 metacharacter escape 7-4
 trap command 7-37
 variable substitution, inhibiting
 7-12

Quoting
See also Quotation mark
 backslash (\) use 7-70
 metacharacter escape 7-4

R

r command
 ed use. *See* ed
 ksh use. *See* Korn shell
 mail use. *See* mail

R command. *See* mail

Random choice
 awk 11-51

Random Choice 11-51

rcp command
 daemon.mn 5-3
 how it works 5-3
 -m option 5-3
 sample command 5-3
 syntax of 5-2
 -u option 5-3, 5-2

Index

read command
 exit status 7-41
 See ed
 See vi
 shell built-in command 7-55
 special shell command 7-41
Read. *See* read command
readonly command
 described 7-41
 shell built-in command 7-55
 special shell command 7-41
records
 multi-line 11-40
Redirection
 argument location 7-9
 case command 7-36
 cd arg command 7-40
 control command 7-36
 diagnostic output 7-7
 file descriptor 7-51
 for command 7-36
 if command 7-36
 minus sign (-) effect 7-50
 pattern matching, use restriction 7-6
 simple command line, appearance
 7-26
 special character, use restriction 7-7
 symbols
 (<), (>) 7-69
 until command 7-36
 while command 7-36
Regular expressions
 awk 11-13
Regular Expressions 11-13
regular expressions 12-1
Regular Expressions)Increasing
 Precedence) 11-56
Regular expressions. *See* ed
rehash command
 C-shell use. *See* C-shell
Relational expressions
 awk 11-12
Relational Expressions 11-12
relative addressing 12-1
Reminder service
 mail 4-24
remote command
 -f option 5-4
 -m option 5-4
 restricting remote execution 5-5
 sample command 5-4
 syntax of 5-4, 5-2, 5-4
Repeat command, vi 2-47
reply command 4-13

Report option. *See* vi
Reserved word, list 7-70
Retrieving files sent with uuto
 See uupick command
Return code 7-16
return command
 shell built-in command 7-55
RETURN key
 bc 6-2

S

s command
 ed use. *See* ed
 mail
 message saving 4-12, 4-8, 4-12,
 4-17, 4-6
-s option
 mail, subject specification 4-5
 shell, invoking 7-54
(ESC
 key
 vi use. *See* vi
scale command 6-9
Scale. *See* bc
Screen size. *See* mail
 screen= option
Screen-oriented commands *See* vi
Scripts
 See ed
 See Shell
Search
 ed use. *See* ed
 vi procedure. *See* vi
Search string. *See* mail, message list
secondary authorizations 10-8
Security
 files 10-13
 passwords 10-12
 practices 10-12
Security Administrator 10-2
sed
 n 12-2
 = function 12-14
 : label function 12-14
 a function 12-6
 addressing 12-4
 b label function 12-14
 B!function 12-13
 c function 12-7
 D function 12-12
 d function 12-6

- scd (*continued*)
 - c 12-2
 - f 12-2
 - flow-of-control 12-2
 - flow-of-control functions 12-13
 - functions 12-6
 - G function 12-13
 - g function 12-9
 - h function 12-12
 - H function 12-13
 - hold and get functions 12-12
 - i function 12-7
 - input/output functions 12-10
 - miscellaneous functions 12-14
 - multiple input-line functions 12-12
 - n 12-2
 - N function 12-12
 - n function 12-6
 - p function 12-10
 - P function 12-12
 - p function 12-9
 - q function 12-15
 - r function 12-11
 - s function 12-8
 - substitution functions 12-8
 - t label function 12-14
 - w function 12-11 , 12-9
 - x function 12-13
- sed command. *See* ed
- Semicolon (;)
 - bc, statement separation 6-22 , 6-4
 - case command break 7-29
 - case delimiter symbol 7-69
 - command list 7-26
 - command separator symbol 7-69
 - C-shell use. *See* C-shell
 - ed use. *See* ed
- Sending files over serial lines
 - See* rcp
- Serial line
 - commands for 5-1
 - telecommunication
 - See* cu command
- set all. *See* vi
- set command
 - C-shell
 - variable value assignment 8-4
 - Korn shell 9-16
 - mail
 - description and use 4-21
 - name-value pair listing 7-21
 - positional parameters, setting 7-11
 - shell built-in command 7-55
 - shell flag, setting 7-19
 - set command (*continued*)
 - special shell command 7-40
 - sh command
 - See also* Shell
 - described 7-1
 - mail 4-15
 - shell, invoking 7-22
 - SHACCT variable 7-14
 - Shell
 - argument passing 7-23
 - awk 11-45
 - command
 - See also* specific command
 - executing while in vi 2-14
 - search procedure 7-3
 - conditional capability 7-28
 - shell
 - cooperation with 11-45
 - Shell
 - creating
 - procedure 7-2
 - described 7-2
 - e option 7-46
 - entering from mail 4-15
 - escape
 - ed procedure. *See* cd
 - execution
 - flag. *See* Shell, option
 - sequence 7-24
 - terminating 7-33
 - exit
 - e option 7-46
 - mail mode return 4-15
 - procedure 7-33
 - t option 7-46
 - function 7-1
 - grammar 7-68
 - in-line input document handling 7-50
 - interactive 7-54
 - interruption procedure 7-36
 - invoking
 - option 7-54
 - procedure 7-22
 - k option 7-46
 - mail
 - shell commands 4-15
 - n option 7-46
 - option
 - See also* specific option
 - description and use 7-46
 - setting 7-19
 - pattern matching facility. *See* Pattern matching facility
 - positional parameter. *See* Positional

Index

Shell (*continued*)

- parameter
- procedure
 - See also* specific shell procedure
 - advantages over C programs 7-45
 - byte access, reducing 7-57
 - creating 7-44
 - described 7-3
 - directory 7-45
 - efficiency analysis 7-56
 - examples 7-60
 - filter, order consideration 7-57
 - option 7-46
 - scripts, examples of 7-60
 - time command 7-55
 - writing strategies 7-55
- redirection ability 7-6
- scripts 7-60
- special command
 - See also* specific special command
 - described 7-40
 - listed 7-40
- special shell variable 7-24
- state 7-18
- t option 7-46
- u option 7-46
- v option 7-19

SHELL
variable 7-14

Shell
variable. *See* Variable
-x option 7-19 , 11-45

Shell facility
awk 11-51

Shell Facility 11-51

.sh_history file. *See* Korn shell

shift command

- argument processing 7-23
- shell built-in command 7-55

Simple Actions 11-8

Simple command. *See* Command

Simple Patterns 11-6

Slash (/)

- bc, division operator symbol 6-5
- command, suppress prepending 7-3
- ed use. *See* ed
- search command. *See* vi

Some Lexical Conventions 11-33

Special character

- See also* Metacharacter
- ed use. *See* ed
- pattern matching facility 7-4

spoofing program 10-13

sprintf 11-24

Standard

- error file. *See* Output
- error output 7-51
- input file. *See* Input
- output file. *See* Output

Star (*)
See also Asterisk (*)
ed metacharacter. *See* ed

Sticky directories 10-11

String

- option. *See* mail
- searching for. *See* vi, searching variable 7-12

String Functions 11-54

Strings and String Functions 11-21

su command 10-6

sub 11-22

Subshell, directory change 7-18

Substitution command. *See* s command

substr 11-24

subsystem authorizations 10-7

Subtraction. *See* bc

Summary

- awk 11-53

System

- awk 11-45
- mailbox. *See* Mailbox

System function
awk 11-45

System Function 11-45

T

t command

- ed use. *See* ed
- t option, shell procedure 7-46

Table command. *See* ed

Tabs

- ed use. *See* ed

tbl command. *See* ed

Telecommunication

- interactive session 5-15 , 5-17
- over serial lines 5-17
- remote terminal
 - See* ct command
 - See* ct command
 - See* cu command
 - See* uucp
 - See* uux command

Temporary file

- trap command 7-38
- use 7-16

term option. *See* vi
 TERM variable 9-14
 terminal authorization 10-8
 terse option. *See* vi
 test command
 argument 7-48
 brackets ([]) used in lieu of 7-47
 description and use 7-47
 operators 7-48
 options 7-47
 shell built-in command 7-55
 Text editor
 ed use. *See* ed
 vi use. *See* vi
 TEXTFILE shell procedure 7-67
 The getline Function 11-40
 The print Statement 11-34
 The printf Statement 11-35
 then clause 7-28
 Tilde escape. *See* mail, compose, escape
 time command 7-55
 Transfer command. *See* ed, t command
 Transferring files
 local site. *See* rcp
 Micnet
 See mail
 See rcp
 phone lines
 See cu command
 See uucp
 See uuto command
 remote site. *See* uucp
 trap command
 description and use 7-36
 multiple traps 7-38
 shell's implementation method 7-38
 special shell command 7-40
 temporary file, removing 7-38
 troff. *See* ed
 Trojan Horse 10-1
 true command 7-50
 trusted computing base 10-2
 Type coercion
 awk 11-57
 type command
 description 7-42
 special shell command 7-42

U

u command
 ed use. *See* ed
 mail 4-14
 See vi
 -u option
 shell procedure 7-46
 ulimit command
 description 7-42
 special shell command 7-42
 umask command
 described 7-42
 shell built-in command 7-55
 special shell command 7-42
 Undo command
 See ed
 See vi
 unset command. *See* mail
 until command
 continue command effect 7-33
 description and use 7-30
 exit status 7-30
 redirection 7-36
 shell built-in command 7-55
 Untrustworthy programs 10-14
 Usage
 awk 11-3
 User
 mailbox. *See* Mailbox
 User-defined functions
 awk 11-32
 User-Defined Functions 11-32
 User-defined variables
 awk 11-8
 User-defined Variables 11-8
 Using awk with Other Commands and
 the Shell 11-45
 /usr/bin directory
 /bin, files duplicated in 7-60
 command search 7-3
 uucp
 abbreviated pathnames 5-9
 advantages of 5-6
 UUCP
 commands 5-6
 uucp
 C-shell considerations 5-9
 dial out site 5-7
 directory permissions 5-7
 disadvantages of 5-6
 file permissions 5-7
 how it works 5-8

Index

uucp (*continued*)

- indirect transfers 5-10 , 5-7
- listing remote UUCP systems 5-7
- m option 5-10
- n option 5-10

UUCP

- networks 5-6

uucp

- options 5-10
- pathnames 5-9

UUCP

- programs 5-6

uucp

- sample command 5-8 , 5-9
- simplest form of 5-8
- status of 5-10
- syntax of 5-8
- transferring files with 5-6

UUCP

- uucp command 5-6
- uuto command 5-6
- uux command 5-6
- when to use 5-1

uname command

- listing remote UUCP systems 5-7

uupick command

- d option 5-13
- how it works 5-12
- m option 5-13
- options 5-13
- quitting 5-13
- retrieving files with 5-12
- sample command 5-13 , 5-12

uustat command 5-10

uuto command

- advantages of 5-6
- disadvantages of 5-6
- how it works 5-12
- public directory 5-12
- retrieving files with uupick 5-12
- sample command 5-12
- syntax of 5-11
- /usr/spool/uucppublic 5-12 , 5-11

uux command

- local site 5-14
- quotation marks 5-14
- quoting the command line 5-14
- remote sites 5-14
- restricting commands 5-13
- sample command 5-14
- security considerations 5-13
- syntax of 5-13
- using 5-13

V

v command

- ed use. *See* ed

- mail 4-4

- v option, printing an input line 7-19

- Value, \$? variable 7-16

Variable

- \$# variable 7-16

- \$! variable 7-17

- assignment

- line command 7-10

- string value 7-12

- bc variable. *See* bc

- command environment, composition 7-20

- conditional substitution 7-52

- described 7-11

- double quotation marks (`__`) 7-12

EDITOR

- See* EDITOR variable

- enclosure 7-13

- execution sequence 7-12

- expansion 7-5

- export 7-16

- expr command 7-49

- file, creating 7-36

- global check 7-46

- HOME. *See* HOME variable

- IFS. *See* IFS variable

- keyword parameter 7-20

- list 7-13

- listing procedure 7-21

- MAIL. *See* MAIL variable

- MAILCHECK. *See* MAILCHECK variable

- MAILPATH. *See* MAILPATH variable

- name defined 7-12

- null value assignment procedure 7-52

- PATH. *See* PATH variable

- positional parameter. *See* Positional parameter

- prefix (\$) 7-12

- PS1. *See* PS1 variable

- PS2. *See* PS2 variable

- set variable defined 7-52

- SHACCT. *See* SHACCT variable

- shell, list of variables 7-13

- SHELL. *See* SHELL, variable

- special variable 7-16

- string value assignment 7-12

- substitution

Variable (*continued*)

- substitution (*continued*)
 - double quotation marks (`__`) 7-12
 - notation 7-70
 - redirection argument 7-6
 - single quotation marks (`'`) 7-12
 - space interpretation 7-13
 - u option effect 7-46
- test command 7-47
- VISUAL
 - See VISUAL variable
- verbose option 9-17
- Vertical bar (`|`)
 - or-if operator symbol (`||`) 7-26
 - pipeline notation 7-7
- vi
 - . command 2-4
 - 0 command
 - cursor movement 2-6
 - abbr command 2-62
 - appending text
 - a command 2-24
 - args command 2-50
 - b command, cursor movement 2-6
 - Bourne shell
 - prompt 2-57
 - breaking lines 2-30
 - buffers
 - delete 2-38
 - naming 2-27
 - selecting 2-27
 - C command 2-35
 - C shell
 - prompt 2-57
 - canceling changes 2-48
 - caret (`^`), pattern matching 2-44 , 2-45
 - cc command 2-35
 - co (copy) command 2-27
 - colon (`:`)
 - line-oriented command, use 2-11
 - status line prompt 2-11
 - command
 - See *also* specific command
 - line-oriented 2-11
 - repeating, using dot (`.`) 2-6
 - screen-oriented 2-11
 - /command
 - searching 2-10
 - Command mode
 - cursor movement 2-5
 - entering 2-3
 - control characters, inserting 2-30
 - copying lines 2-27
 - correcting mistakes 2-25

vi (*continued*)

- crash, recovery from 2-55
- C-shell
 - TERM variable 2-58
 - terminal type, setting 2-58
- Ctrl-b
 - scrolling 2-6
- Ctrl-d
 - scrolling 2-6
 - subshell exit 2-55
- Ctrl-f
 - scrolling 2-6
- Ctrl-g
 - file status information 2-11 , 2-54
- Ctrl-j
 - inserting 2-30
- Ctrl-l
 - screen redraw 2-55
- Ctrl-q
 - inserting 2-30
- Ctrl-r
 - screen redraw 2-55
- Ctrl-s
 - inserting 2-30
- Ctrl-u
 - deleting an insert 2-32
 - scrolling 2-6
- Ctrl-v
 - use 2-30 , 2-62
- current file (`%`) 2-62
- current line
 - deleting 2-31 , 2-6
 - designated 2-3
 - line containing cursor 2-4
 - number, finding out 2-27
- cursor movement
 - + key 2-21
 - \$ key 2-22
 - B command 2-20
 - backward 2-22
 - BKSP 2-19
 - character 2-19
 - Ctrl-n 2-22
 - Ctrl-p 2-22
 - down 2-19 , 2-5
 - e command 2-20
 - end of file 2-5
 - f command 2-19
 - file
 - end 2-5
 - forward 2-21
 - h command 2-19
 - H command 2-22
 - j 2-22

Index

vi (continued)

cursor movement (continued)

- j command 2-19
 - k command 2-19, 2-22
 - keys 2-5
 - l command 2-19
 - L command 2-22
 - left 2-19, 2-20, 2-5
 - line
 - beginning 2-6
 - end 2-6
 - number 2-5, 2-21
 - LINEFEED key 2-21
 - lower left screen 2-5
 - M command 2-22
 - number of specific line 2-5
 - pattern search 2-10
 - right 2-19, 2-20, 2-5
 - (Return
 - key 2-21
 - screen 2-22
 - scrolling 2-22, 2-6
 - SPACE 2-19
 - T command 2-20
 - up 2-19, 2-5
 - upper left screen 2-5
 - w command 2-20
 - word
 - backward 2-6
 - forward 2-6, 2-20
- cw command 2-34
- d\$ command 2-6
- d0 command 2-6
- date, finding out 2-14
- dd command 2-31, 2-6
- delete buffer
 - use 2-38
- deleting text
 - by character 2-30
 - by line 2-31
 - by word 2-31
- D 2-31
- dd command 2-31, 2-6
- deleting an insert 2-32
- dw command 2-31
- methods 2-6
- repeating a delete 2-47
- undoing a delete 2-46, 2-5
- X command 2-30
- demonstration 2-2
- described 2-1
- dollar sign (\$) *(continued)*
 - cursor movement 2-6
 - pattern matching 2-44

vi (continued)

- dollar sign (\$) *(continued)*
 - use in line address 2-32
- dot (.)
 - command 2-6
 - use in line address 2-32
- dw command 2-6
- editing several files
 - changing the order 2-51
- end-of-line
 - displaying 2-59
- entering vi
 - filename specified 2-18
 - line specified 2-18
 - procedure 2-2
 - several filenames 2-49
 - word specified 2-19
- error messages
 - brevity 2-60
 - turning off 2-53
- ESC key 2-62
- Escape key, Insert mode exit 2-3, 2-55
- exclamation point (!)
 - shell escape 2-14
- EXINIT environment variable 2-57
- exiting
 - :q! 2-16
 - saving changes to file 2-13, 2-48
 - temporarily 2-14, 2-52
 - without saving changes 2-48
 - :x command 2-16, 2-48
 - ZZ command 2-48
- .exc file 2-63
- file
 - creating 2-2
 - exit without saving, :q! 2-16
 - saving 2-16
 - status information display 2-10
 - status information procedure 2-11
- filename
 - finding out 2-54
 - planning 2-50
- G command
 - cursor movement 2-5
- global substitution
 - command syntax 2-42
- goto command 2-5
- H command
 - cursor movement 2-5
- i command
 - inserting text 2-3
- ignorecase option 2-41, 2-59
- Insert command 2-24, 2-3

vi (continued)

- Insert mode
 - entering 2-3
 - exiting 2-3
- inserting text
 - beginning of line 2-24
 - commands 2-24
 - control characters 2-30
 - from another file 2-14
 - from other files 2-14 , 2-25 , 2-26
- i command 2-24
- Insert mode 2-3
 - repeating an insert 2-25 , 2-47
 - See vi, appending text
 - undoing an insert 2-46 , 2-5 , 2-55
- invoking 2-18 , 2-19 , 2-2 , 2-49
- j command
 - cursor movement 2-5
- J command 2-30
- joining lines 2-30
- k command
 - cursor movement 2-5
- l command
 - cursor movement 2-5
- leaving
 - See vi, exiting
 - See vi, quitting
- line addressing
 - dollar sign 2-32
 - dot (.) 2-32
 - procedure 2-31
- line numbers, displaying
 - linenumber option 2-15 , 2-59
 - nu command 2-27 , 2-54
- line-oriented commands
 - :args 2-50
 - colon (:) use 2-11
 - deleting text 2-31
 - :e 2-26 , 2-51
 - :e# 2-52
 - entering 2-11
 - :f 2-54
 - :file 2-54
 - mode 2-53
 - moving text 2-36
 - :n 2-50
 - nu 2-27 , 2-54
 - :q 2-48
 - :r 2-25
 - :rew 2-51
 - :s 2-35
 - status line, display 2-10
 - :w 2-26
 - :wq 2-48

vi (continued)

- list option 2-59
- .login file
 - terminal type, setting 2-58
- magic option 2-45 , 2-61
- mail, entering vi from compose mode 4-4
- map command 2-62
- marking lines 2-26
- mesg option 2-61
- mistakes, correcting 2-25
- mode
 - Command mode 2-55
 - determining 2-55
 - Insert mode 2-55
- moving text 2-36
- n command 2-10 , 2-41
- new line, opening 2-25
- next command 2-50
- number option 2-59
- opening a new line 2-25
- options
 - displaying 2-58
 - ignorecase 2-41
 - ignorecase option 2-59
 - linenumber option 2-27
 - list 2-16
 - list option 2-59
 - magic option 2-45 , 2-61
 - mesg option 2-61
 - number option 2-36 , 2-59
 - report option 2-60
 - setting 2-57 , 2-59
 - term option 2-60
 - terse option 2-60
 - warn option 2-53 , 2-61
 - wrapscreen option 2-41 , 2-61
- overstrike commands 2-32
- pattern matching
 - beginning of line 2-44
 - caret (^) 2-45
 - character range 2-45
 - end of line 2-44
 - exceptions 2-45
 - special characters 2-45
 - square brackets ([]) 2-45
 - percent sign, current file 2-62
- period (.)
 - See also vi, dot (.)
 - pattern matching 2-44
- problem solving 2-55
- .profile file
 - terminal type 2-57
- putting 2-27

Index

vi (continued)

- :q! 2-16
- Q command 2-53
- quitting
 - See also* vi, exiting
 - 2-14, 2-16, 2-48, 2-52, 2-55
 - "
- r command 2-14, 2-32, 2-33
- read command 2-14
- redrawing the screen 2-55
- Repeat command 2-47
- repeating a command 2-47
- replacing
 - line 2-35
 - word 2-34, 2-35
- report option 2-60
- rew command 2-51
- s command 2-34
- saving a file 2-49
- screen, redrawing 2-55
- screen-oriented commands 2-11
- scrolling
 - backward 2-6
 - down 2-23, 2-6
 - forward 2-6
 - up 2-22, 2-6
- search and replace
 - c option 2-43
 - choosing replacement 2-43
 - command syntax 2-42
 - global
 - warning 2-47, 2-42
 - p option 2-43
 - printing replacement 2-43
 - word 2-42
- searching
 - See also* vi, search and replace
 - backward 2-40
 - caret (^) use 2-44, 2-45
 - case significance 2-41, 2-59
 - dollar sign (\$) 2-44
 - forward 2-10, 2-40
 - next command 2-41
 - period (.) 2-44
 - procedure 2-10
 - repetition 2-10
 - slash (/) 2-10
 - special characters 2-40, 2-61
 - square brackets ([]) 2-45
 - status line, display 2-10
 - wrap 2-10, 2-41, 2-61
- session, canceling 2-16
- set all, option list 2-16
- set command 2-16, 2-57, 2-58

vi (continued)

- setting options 2-16, 2-57, 2-59
 - shell
 - command, executing 2-14
 - escape 2-52
 - slash (/)
 - search command delimiter 2-10
 - special characters
 - matching 2-45
 - searching for 2-40, 2-61
 - vi filenames 2-49
 - status line
 - line-oriented command entry 2-11
 - location 2-10
 - prompt, colon (:): use 2-11
 - string
 - pattern matching 2-45
 - searching for 2-10
 - subshell
 - exiting 2-55
 - substitute commands 2-34
 - switching files 2-51
 - system crash
 - file recovery 2-56
 - tabs
 - displaying 2-59
 - TERM variable
 - Bourne shell 2-57
 - termcap 2-57
 - terminal type, setting
 - Bourne shell 2-57
 - C-shell 2-58
 - instructions 2-60
 - terse option 2-60
 - time, finding out 2-14
 - u command 2-4, 2-46, 2-55
 - Undo command 2-4
 - w command, cursor movement 2-6
 - warn option 2-53, 2-61
 - warnings, turning off 2-61
 - word, deleting 2-6
 - wrapscreen option 2-41, 2-61
 - write messages 2-61
 - writing out a file
 - :wq command 2-48, 2-49
 - :x command 2-16, 2-48
 - x command 2-6
 - yanking lines 2-26, 2-29
 - ZZ command 2-48
- vi option 9-17
- vi, used in mail
 - entry from command mode 4-4
- vi -x 10-16
- visual command. *See* mail

VISUAL variable 9-14 , 9-3

W

w command
 ed use. *See* ed
 mail 4-17 , 4-6
 vi use. *See* vi
 wait command
 described 7-43
 shell built-in command 7-55
 special shell command 7-43
 warn option. *See* vi
 while command
 break command effect 7-32
 continue command effect 7-32
 description and use 7-30
 exit status 7-30
 loop 7-61
 redirection 7-36
 shell built-in command 7-55
 test command 7-47
 Word
 grammar 7-69
 Word Frequencies 11-50
 wrapscan option. *See* vi
 Write out. *See* w command
 WRITEMAIL shell procedure 7-67

X

x command
 mail
 exit 4-17
 session abort 4-17
 vi use. *See* vi
 -x option, printing a command 7-19
 XENIX command
 directory residence
 C-shell 8-4

Z

z command
 vi scroll 2-23
 ZZ command 2-48



Change Information

This is a summary of the changes that have been made to the previous version of this manual. The chapters, page numbers, and/or paragraphs mentioned in this summary refer to the previous manual.

Title: Altos UNIX System V/386 Release 3.2 User's Guide

Revised Part Number: 690-23408-002

Previous Part Number: 690-23408-001

Date: March 1991

Changes:

Deleted the phrase "for the 486" from the title of this book to indicate that non-486 machines can run this operating system.

The "Guide to Your Altos UNIX System V/386 Release 3.2 Documentation" and the "Operating System Documents for Different Audiences" pages located in the front matter of this book were both changed to identify which manuals are automatically shipped with every run-time system, and which manuals are available as spare parts.

Inserted a new chapter, "The Korn Shell," as Chapter 9. The previous chapters nine through 13 were incremented accordingly (i.e., the previous Chapter 9 is now Chapter 10, etc.).

Minor editing changes were made throughout this manual.

Changed the following pages:

Page	Description
4-10	Fixed header display. Also, the command prompt for mail is now an ampersand (&) not a question mark (?).
4-11	

P/N 690-23408-002

Printed in U.S.A.

5/91



Altos Computer Systems

2641 Orchard Parkway, San Jose, CA 95134
408/432-6200, FAX 408/435-8517