Stardent

START HERE: USERS GUIDE

Stardent Computer Inc. Part Number: 340-0118-01

CONTENTS

Preface

1 Overview

System Interface Hardware 1-1
Monitor 1-1
Keyboard 1-2
Mouse and Mouse Pad 1-3
Junction Box 1-3
Operating System 1-3
Kernel 1-4
File System 1-4
Shell 1-5
Commands 1-6
How to Execute Commands 1-6
How Commands Are Executed 1-8
Typing Conventions 1-9
Correcting Typing Errors 1-9
Using Special Characters as Literal Characters 1-10
Typing Speed 1-10
Logging In 1-11
Password 1-11
X Window System 1-13
Simple Commands 1-14
Logging Off 1-15

2 Display

Contents 2-1
Introduction 2-1
A. Session One 2-1
B. Basics 2-2
Using the Mouse 2-4

Opening and Closing Windows 2-6
Resizing Windows 2-7
Moving Windows 2-7
Raising, Lowering, and Circulating Windows 2-8
The System Menu 2-8
The User Menu 2-10
Modifying the Default Window Configuration 2-11
Modifying the Default Mouse Keys 2-11
C. Resources 2-12
General Literature 2-12
D. Quick Reference 2-13

3 UNIX Essentials

Contents 3-1
Introduction 3-1
A. Session One 3-2
B-1. Basics: The File System 3-2
How the File System is Structured 3-2
Your Place in the File System 3-3
Your Home Directory 3-3
Your Current Directory 3-4
Pathnames 3-5
Full Pathnames 3-5
Relative Pathnames 3-6
Naming Directories and Files 3-9
Organizing a Directory 3-10
Creating Directories: the **mkdir** Command 3-10
Listing the Contents of a Directory: the **ls** Command 3-11
Listing All Names in a File 3-13
Listing Contents in Short Format 3-14
Changing the Current Directory 3-15
Removing Directories: the **rmdir** Command 3-17
Accessing and Manipulating Files 3-18
Concatenate and Print Contents of a File 3-19
Paging Through the Contents of a File 3-20
Print Partially Formatted Contents of a File 3-23
Requesting a Paper Copy of a File: the **lp** Command 3-24
Making a Duplicate Copy of a File: the **cp** Command 3-25
Moving or Renaming a File: the **mv** Command 3-26
Removing a File: the **rm** Command 3-28
Counting Lines, Words, and Characters in a File 3-29
Protecting Your Files: the **chmod** Command 3-30
How to Determine Existing Permissions 3-31
A Note on Permissions and Directories 3-36

Identifying Differences Between Files: **diff** 3-36
Searching a File for a Pattern: the **grep** Command 3-37
Sorting and Merging Files: the **sort** Command 3-38
B-2. Basics: The Shell 3-39
Shell Command Language 3-40
Pattern Matching 3-40
Special Characters 3-43
Input and Output Redirection 3-46
Redirecting Input: the < Sign 3-46
Redirecting Output to a File: the > Sign 3-47
Appending Output to an Existing File: the >> Symbol 3-47
Useful Applications of Output Redirection 3-48
Combining Input and Output Redirection 3-48
Combining Background Mode and Output Redirection 3-49
Supplying Lines of Input to a Command 3-49
Redirecting Output to a Command: the Pipe (|) 3-50
Substituting Output for an Argument 3-51
Executing and Terminating Processes 3-52
Running Commands at a Later Time: **batch** and **at** 3-52
Obtaining the Status of Running Processes 3-54
Terminating Active Processes 3-55
Using the **nohup** Command 3-55
Shell Programming 3-56
Creating and Executing a Simple Shell Program 3-56
Creating a **bin** Directory for Executable Files 3-57
Warnings about Naming Shell Programs 3-58
Variables 3-58
Positional Parameters 3-58
Special Parameter: $ 3-59
Special Parameter: $* 3-60
Named Variables 3-61
Using the **read** Command 3-63
Substituting Command Output for Value of Variable 3-65
Assigning Values with Positional Parameters 3-65
Shell Programming Constructs 3-66
Comments 3-66
Return Codes 3-67
Looping With the **for** Loop 3-67
Looping With the while - do Loop 3-70
The Shell's Garbage Can: **/dev/null** 3-71
if...the Conditional Constructs 3-72
if...then...else Conditional Constructs 3-73
The **test** Command for Loops 3-74
Using the Test Command With Return Codes 3-75
case..esac Conditional Constructs 3-75
Unconditional Control Statements 3-78

Debugging Programs 3-78
The C-Shell 3-81
Using the C-Shell 3-82
The C-Shell History Feature 3-83
The C-Shell Alias Feature 3-84
Modifying Your Login Environment 3-84
Modifying Your *.profile* 3-85
An Example .profile 3-85
Setting Terminal Options 3-86
Using Shell Variables 3-86
Modifying Your C-Shell Environment: The *.login* 3-89
C. Resources 3-90
General Literature 3-90
D. Quick Reference 3-92

4 The *vi* Editor

Contents 4-1
Introduction 4-1
A. Session One 4-2
B. Basics 4-2
Getting Started 4-3
Terminal Name 4-3
Creating a File 4-4
Creating Text: The Input Mode 4-5
Leaving Input Mode 4-6
Editing Text: the Command Mode 4-6
Basic Cursor Movement Commands 4-6
Deleting Text 4-7
Adding Text 4-8
Quitting vi 4-9
Moving the Cursor Around the Screen 4-10
Moving the Cursor to the First/Last Character of a Line 4-11
Move the Cursor to a Specific Character on a Line 4-12
Moving the Cursor Line by Line 4-12
Moving the Cursor Word by Word 4-13
Moving the Cursor Sentence by Sentence 4-14
Moving the Cursor Paragraph by Paragraph 4-15
Moving the Cursor Within the Window 4-15
Moving the Cursor Outside the Window 4-16
Scrolling the Text 4-16
Moving to a Specified Line 4-17
Line Numbers 4-17
Searching for a Pattern of Characters: / and ? 4-18
Creating Text 4-20

Deleting Text 4-21
Deleting Text in Input Mode 4-21
Undoing the Last Command 4-21
Deleting Text in Command Mode 4-21
Modifying Text 4-22
Replacing Text 4-22
Substituting Text 4-23
Changing Text 4-24
Cutting And Pasting Text 4-25
Moving Text 4-25
Fixing Transposed Letters 4-26
Copying Text 4-26
Copying or Moving Text Using Registers 4-27
Other Commands 4-28
Repeating the Last Command 4-28
Joining Two Lines 4-29
Clearing and Redrawing the Window 4-29
Changing Cases 4-29
Using Line Editing Commands in *vi* 4-30
Temporarily Returning to the Shell 4-30
Saving Changes or Writing Text to a New File 4-31
Finding the Line Number 4-31
Deleting the Rest of the Buffer 4-32
Adding a File to the Buffer 4-32
Global Substitution 4-32
Quitting vi 4-33
Special Options For vi 4-34
Recovering a File Lost by an Interrupt 4-34
Editing Multiple Files 4-35
Viewing a File 4-35
C. Resources 4-35
General Literature 4-36
D. Quick Reference 4-36

5 Communication

Contents 5-1
Introduction 5-1
A. Session One 5-2
B. Basics 5-2
Sending Mail to One Person 5-2
Undeliverable Mail 5-4
Sending Mail to Several People Simultaneously 5-5
Sending Mail to Remote Systems 5-5
Managing Incoming Mail 5-7

Sending and Receiving Files Via the **mail** Command 5-11
Networking 5-12
Ethernet/Cheapernet: the **rlogin** Command 5-13
Ethernet/Cheapernet: the **rcp** Command 5-13
Ethernet/Cheapernet: the **rsh** Command 5-14
RS-232 Port Communications: the **ct** Command 5-14
RS-232 Port Communications: the **cu** Command 5-15
C. Resources 5-18
General Literature 5-18
D. Quick Reference 5-18

6 Getting Started in Programming

Contents 6-1
Introduction 6-1
A. Session One 6-2
B. Basics 6-2
Compiling 6-2
Command Line Options 6-3
Preprocessor Options 6-3
Compiler Options 6-3
Loader Options 6-4
Compilation Control Statements 6-4
Compiler Directives 6-5
The Stardent 1500/3000 Debugger 6-5
dbg Tasks 6-6
Debugging Tools 6-7
Linker 6-8
Archive Libraries 6-8
Code Optimization 6-8
C. Resources 6-9
D. Quick Reference 6-10
Compiler Optimization 6-11
Fortran Compiler Options 6-11
C Compiler Options 6-13
C Preprocessor Options 6-14
Loader Table Options 6-14

7 Example Sessions

A Sample Session 7-2
Sample Application 7-3
Creating The New Windows 7-3
Creating The Sample Directory 7-5

Editing a Program 7-5
Compiling The Program 7-6
Running The Program 7-7
Running Under Control Of The Debugger 7-7
A Sample X Program 7-10

A Inventory of Documentation

System Documentation A-1
UNIX A-1
General Literature A-1
X Window System A-2
General Literature A-2
Communications A-2
General Literature A-2
Programming A-3
General Literature A-3
Graphics A-3

PREFACE

Start Here is the manual to use when you first work with the Stardent 1500/3000 operating system. Its purpose is to give you basic information for its tools.

Start Here is intended for those who are not familiar with the UNIX operating system, X Window, the *vi* editor, etc. We assume that you do have prior mainframe or PC experience; this book is **not** a primer for first-time computer users.

Contents

After the introductory **Chapter 1: Overview**, there are five chapters, each dealing with a distinct group of tools — the X Window System, UNIX essentials, the *vi* editor, communications, and getting started in programming — from four different approaches:

- Each tool group is introduced with **A. Session One**, which provides a few commands so that you can start working.
- The next section is **B. Basics**, a group of more advanced commands, along with brief explanations.
- **C. Resources** points at additional information available from Stardent and elsewhere.
- Finally, **D. Quick Reference** is a list of the most often used commands for each tool group.

After the "vertical" description of distinct tool groups, **Chapter 7: Example Sessions**, provides "horizontal" information, with true-to-life, across-applications sequences such as starting X, creating a file with *vi*, closing the file, compiling and running the program.

A matrix of the structure looks like this:

	A. Session One	B. Basics	C. Resources	D. Quick Ref
Ch. 2 Display (X)	2-A	2-B	2-C	2-D
Ch. 3 UNIX Essentials	3-A	3-B1 & 3-B2	3-C	3-D
Ch. 4 vi Editor	4-A	4-B	4-C	4-D
Ch. 5 Communications	5-A	5-B	5-C	5-D
Ch. 6 Programming	N/A	6-B	6-C	6-D

Material in this guide is presented as briefly as possible so that instead of spending time with reading, you get quick (if not-too-dirty) solutions to initial problems you're certain to encounter when first using any system.

While Stardent 1500/3000 provides a great deal of functionality and flexibility, this manual deals with a limited set of commands only. See the *Commands Reference Manual* for complete information.

Again, here is what you will find in this guide:

- Chapter 1: *Overview*, is a description of the end-user interface, and instructions on logging in.
- Chapter 2: *Display*, is a guide to Stardent 1500/3000's X Window System.
- Chapter 3: *UNIX Essentials*, is a beginner's guide to the UNIX V.3 operating system, with emphasis on the shell command language, pathnames, directory and file structure.
- Chapter 4: *The vi Editor*, is a guide to the standard UNIX *vi* editor, on using *vi* to create and edit files.
- Chapter 5: *Communications*, is about exchanging information with others on the network, both through electronic mail and the transfer of files.
- Chapter 6: *Getting Started in Programming*, provides information so that you may begin using Stardent 1500/3000's programming tools.
- Chapter 7: *Example Sessions* provides several sequences of actual work with Stardent 1500/3000.

- Appendix A: *Inventory of Documentation,* is a roadmap to all Ardent documentation as well as to some outside literature.

OVERVIEW

CHAPTER ONE

This chapter is an overview of Stardent 1500/3000's end-user interface followed by instructions on logging in. The rest of the guide provides information about the interface.

Stardent 1500/3000's end-user interface consists of

- System interface hardware
- UNIX operating system
- X Window system

System Interface Hardware

The system interface hardware includes the monitor, keyboard, mouse and mouse pad, any optional input devices you have configured to your Stardent 1500/3000, and the junction box. The following paragraphs describe these devices.

Monitor

The Stardent 1500/3000 monitor has a 19" (diagonal measure) screen that provides 1280 by 1024 pixel resolution. The monitor has a self-contained power supply and operates from a standard 120V, 60Hz wall socket.

To configure the monitor, use the Stardent 1500/3000 *Administration/Installation Guide*.

Keyboard

Stardent 1500/3000's keyboard contains a full complement of ASCII and special function keys in an industry-standard layout. Figure 1-1 is a diagram of the keyboard. The keys correspond to

- Letters of the English alphabet (both upper case and lower case)
- Numerals 0 through 9
- A variety of symbols (including ! @ # $ % ^ & ( ) _ – + = ~ ' { } [ ] \ : ; " ' < > , ? /)
- Specially defined functions (such as BREAK), also abbreviated (such as CTRL for control and ESC for escape).

Mouse and Mouse Pad

Stardent 1500/3000 includes an optical mouse with a high resolution, 200 lines per inch mouse pad. Use the mouse to manipulate the windows and icons on the screen. See the *X Window System* section in this chapter and Chapter 2: *Display* for more information.

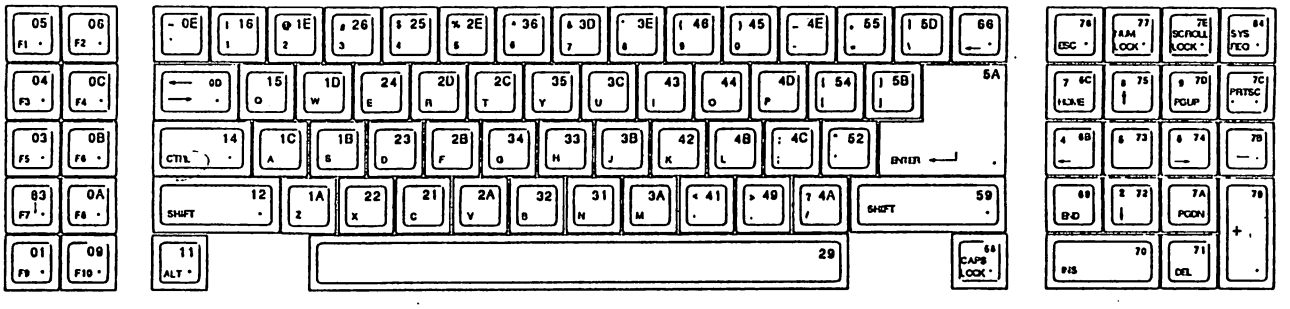


Figure 1-1. Keyboard Layout

Junction Box

The junction box supplies connectors for the monitor, mouse, and keyboard as well as for other optional devices, such as graphics tablet, knob box, light pen, joy sticks, and trackballs. While the junction box must be close to your monitor, it can be as far as 200 feet away from the system module (the cabinet that houses the Stardent 1500/3000 processors and memory).

Operating System

The Stardent 1500/3000 operating system is a set of programs (or software) that controls the Stardent 1500/3000 computer, acts as the link between you and the computer, and supplies tools to help you do your work. It includes a full implementation of AT&T's UNIX System V Release 3, as well as features from Berkeley 4.3 UNIX, high-performance enhancements, and integrated graphics support.

The Stardent 1500/3000 operating system provides an uncomplicated, efficient, and flexible working environment, allowing you to

- Communicate with Stardent 1500/3000 and receive responses
- Perform a wide variety of tasks: running programs, editing files, using special software capabilities such as graphics
- Execute more than one program simultaneously

The operating system has four major components:

- kernel
- file system
- shell
- commands

Kernel

The kernel controls access to Stardent 1500/3000, manages Stardent 1500/3000's memory, maintains the file system, and allocates Stardent 1500/3000 resources among users.

File System

The file system is a logical method of organizing, retrieving, and managing information. The structure of the file system is hierarchical; if you could see it, it might look like an organization chart or an inverted tree. The file is the basic unit of the file system and it can be any one of three types: an ordinary file, a directory, or a special file. Figure 1-2 is a representation of the file system. The large rectangles represent directories and the small rectangles represent ordinary or special files.

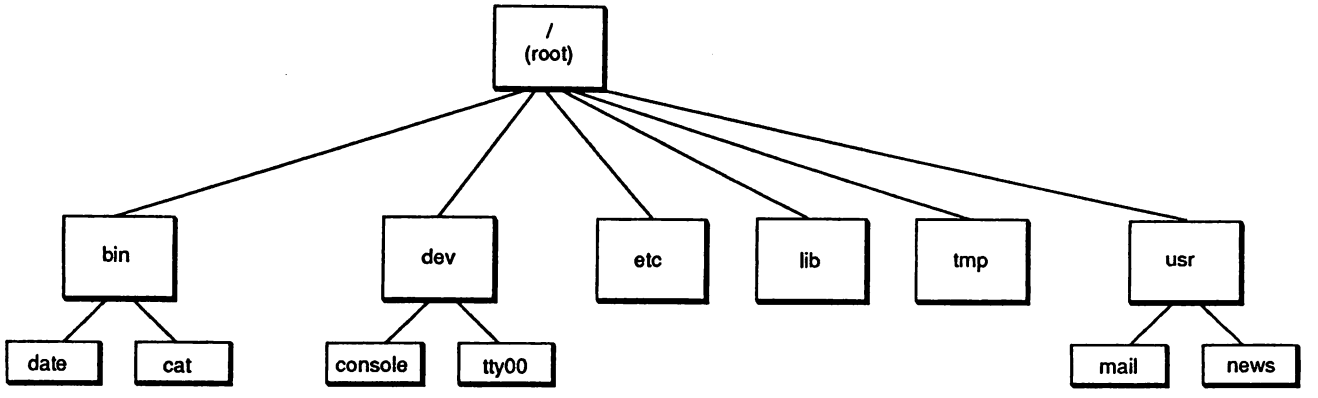


Figure 1-2. Hierarchical Structure of the File System

Shell

The shell is a command interpreter that allows you to communicate with the operating system. The shell reads the commands you enter and interprets them as requests to execute programs, access files, or provide output. The shell is also a powerful programming language, not unlike the C programming language, providing conditional execution and control flow features.

The system features both the Bourne shell and the more contemporary Berkeley C-shell, which:

- Maintains a history of recently executed commands, and includes shorthand ways of modifying or reissuing them.
- Provides a simple way to modify or redefine commands (the **alias** feature).
- Allows stopping and restarting processes, and move jobs from the background to the foreground (and vice versa).

Chapter 3 offers a description of the C-shell.

NOTE
Throughout this guide, the term "shell" is used to refer to the standard UNIX (Bourne) shell. All references to the C-shell are explicit.

Commands

Programs that can be executed by Stardent 1500/3000 without need for translation are called executable programs or commands. This guide describes many of the commands you will use on a regular basis. If you need additional information on these or other commands, refer to the *Commands Reference Manual.*

NOTE
The notation ↵ is used throughout this manual as an instruction to press the carriage return key.

All of the commands in the *Commands Reference Manual* are available online. Use the **man** (short for manual page) command to print a description of any command. For example, to print a description of the **date** command, type

man date ↵

Your request and the system's response looks like this on your screen.

```
Stardent 1500/3000: man date
DATE(1)                    USER COMMANDS                    DATE(1)

NAME
        date - display or set the date

SYNOPSIS
        date [ -u ] [ yymmddhhmm [ .ss ] ]

DESCRIPTION
        date displays the current date and time when used without an
        argument

        Only the super-user may set the date.  yy is the last two
    .
    .
    .
```

How to Execute Commands

To make your requests comprehensible to the operating system, you must present each command in the correct command line syntax. This syntax defines the order in which you enter the components of a command line. Correct syntax is essential for the shell to understand and interpret your request.

The syntax of a typical command line looks like this.

command option(s) argument(s) ↵

command is the name of the program you want to run.

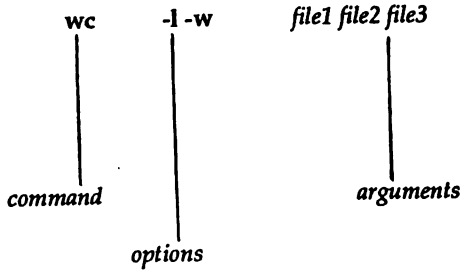
option modifies how the command runs.

argument specifies data on which the command is to operate (usually the name of a directory or file).

On every command line you must type at least two components: the command name and the carriage return key ↵. A command line may also contain either options or arguments, or both.

In command lines that include options and/or arguments, the component words are separated by at least one blank space. If an argument name contains a blank, enclose that name in single or double quotation marks. For example, if the argument to your command is **sample 1**, you can type it as follows: **"sample 1"**. If you forget the quotation marks, the shell interprets **sample** and **1** as two separate arguments.

Some commands allow you to specify multiple options and/or arguments on a command line. Consider the following command line:



In this example, **wc** is the name of the command and two options, **–l** and **–w**, are specified. (The operating system usually allows you to group options such as these to read **–lw**, if you prefer.) In addition, three files (*file1*, *file2*, and *file3*) are specified as arguments. Although most options can be grouped together, arguments cannot.

The following examples show the proper sequence and spacing in command line syntax:

Incorrect	Correct

wc*file*	wc *file*
wc–l*file*	wc –l *file*
wc –l w *file*	wc –lw *file* or wc –l –w *file*
wc *file1file2*	wc *file1 file2*

Remember, regardless of the number of components, you must end every command line by pressing the ↵ key.

How Commands Are Executed

Figure 1-3 shows the flow of control when the operating system executes a command.

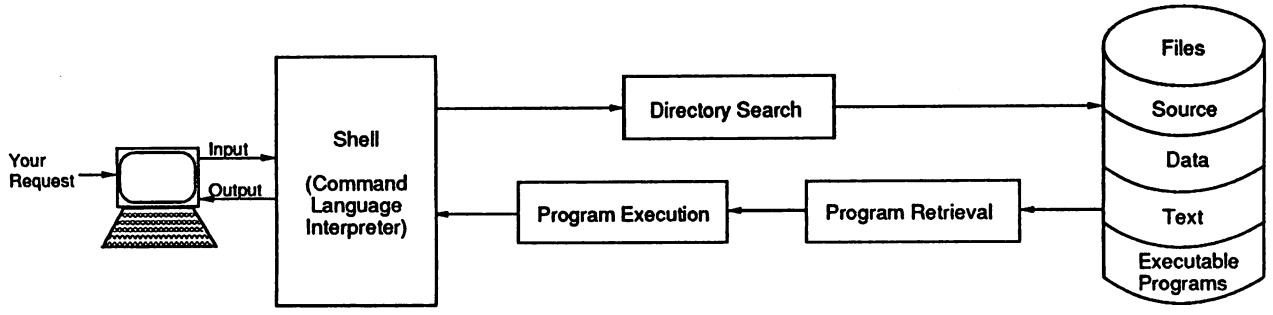


Figure 1-3. Command Execution

To execute a command, you enter a command line when a prompt (such as Stardent 1500/3000:) appears on your screen. The shell considers your command as input, searches through one or more directories to retrieve the program you specified, and conveys your request along with the program requested to the kernel. The kernel then follows the instructions in the program and executes the command you requested. When the program finishes running, the shell signals that it is ready for your next command by printing another prompt.

Typing Conventions

The operating system requires that you enter commands in lower case letters (unless the command includes an upper case letter). You can perform certain tasks, such as erasing letters or deleting lines, simply by pressing one key or entering a specific combination of special characters. A list of some special character combinations follows.

⌫ (backspace)	Erases a character.
CTRL d	Stops input to the system or logs you off.
CTRL s	Temporarily stops output from printing on the screen.
CTRL q	Makes the output resume printing on the screen after it has been stopped by the CTRL s command.
CTRL u	Kills the current command line.
CTRL c	Stops execution of a program or command.
↵	Ends a line of typing and puts the cursor on a new line.

NOTE
Characters preceded by the letters CTRL are called control characters. To type a control character, hold down the control key and press the specified letter.

Correcting Typing Errors

To erase a single character, use the ⌫ (backspace) key. When you press backspace, the cursor backs up over your errors, erasing them as it goes. When you erase an error with the ⌫ key, the line of text on the screen looks as though it was typed perfectly.

If you have started typing a command line and then change your mind, press CTRL u. The command is cancelled and a system prompt appears. If you want to stop execution of a program that is currently running, press CTRL c. The program stops running and the system prompt appears. Here are a couple of examples.

```
Stardent 1500/3000: date^u
Stardent 1500/3000:
```

```
Stardent 1500/3000: wc file1↵
CTRL C
Stardent 1500/3000:
```

Using Special Characters as Literal Characters

If you want to use a special character and assign it its literal meaning you must tell the system to ignore the character's special meaning. The backslash (\) enables you to do this. Type a \ before any special character that you want to have treated as it appears. By doing this you essentially tell the system to ignore this character's special meaning and treat it as a literal unit of text.

For example, suppose you want to add the following sentence to a file:

> Only one # appears on this sheet of music.

The shell interprets the special character # as a request to delete a character. To prevent the operating system from using this interpretation, enter a \ in front of the #. If you do not, the system erases the space after the word one and prints your sentence as follows:

> Only one appears on this sheet of music.

To avoid this, type your sentence as follows:

> **Only one \# appears on this sheet of music.**

The following are special characters understood by the shell.

```
? @ # $ ^ & * ( ) ` [ ] \  |  ; ' " < >
```

Typing Speed

After the prompt appears on the screen, you can type as fast as you wish, even when the operating system is busy executing a command. Because your input and the system's output the printout on the screen may appear garbled. While this may be inconvenient, it does interfere with the operating system, which has read-ahead capability: input and output are handled separately.

The system takes and stores input (your next request) while it sends output (its response to your last request) to the screen.

Logging In

Before you can log in, the Stardent 1500/3000 must be installed and registered. If this is not done yet, see the *Installation/Administration Guide* for instructions.

If your Stardent 1500/3000 is installed and if you have a login name and password, you are ready to login. Turn on your Stardent 1500/3000. When the login: prompt appears, type your login name and press ↵. For example, if your login name is *starship*, your login line looks like this:

```
login: starship↵
```

Remember to type in lower case letters. If you use upper case from the time you log in, the operating system expects all input in upper case and responds in upper case exclusively until the next time you log in. The operating system accepts and runs many commands typed in upper case, but does not allow you to edit files while in upper case mode.

Password

Next, the system prompts you for your password. Type your password and press ↵. For security reasons, the operating system does not print (or echo) your password on the screen.

If both your login name and password are correct, the system prompt appears on your screen. The entire procedure looks like this:

```
login: starship↵
password: your-password↵
Stardent 1500/3000:
```

You may see some system messages between the password line and the system prompt. In any event, the system prompt is your signal that Stardent 1500/3000 is ready to accept your commands.

For security reasons it is a good idea to choose your own private password after you login for the first time. To do so, issue the **passwd** command. The system prompts you first to enter your old password, then to enter the new password of your choice, and finally to confirm the new password by entering it again. This is

what the procedure looks like on your screen.

```
Stardent 1500/3000: passwd↵
Changing password for starship ↵
Old password: your_old_password↵
New password: your_new_password↵
Re-enter new password: your_new_password↵
Stardent 1500/3000:
```

Note that there are several restrictions on passwords.

- They must have at least six characters, and only the first eight characters count.
- There must be at least two alphabetic characters (upper or lower case) and one numeric or special character.
- For security reasons you should not use a variation on your login name as your password. Certain variations are in fact illegal.
- New passwords must differ from old passwords by at least three characters.

Here are some legal passwords:

9754Hi *&whatis* *zz6llffpp (the last p is ignored)*

If you make a typing mistake when logging in, the operating system prints the message

login incorrect

on your screen. Then it gives you a second chance to log in by printing another login: prompt.

```
login: ttarship↵
password: your-password↵
login incorrect
login:
```

If you have any problems logging in please see the *Installation Guide.*

When you log in, you see the screen is divided into windows or work areas. Think of the screen as a desktop, and the windows on the screen as sheets of paper that you can move around, stack, file, and so on. You can run programs, edit files, and send mail messages within each window, just as when the screen has only one window. You can also tailor your windows and icons to suit you. Figure 1-4 is a picture of a monitor screen with the default windows and icons displayed.

The mouse controls the mouse pointer. If you move the mouse around on the mouse pad right now, notice how the mouse pointer moves around the screen. Also notice that the mouse pointer changes shape. When the mouse pointer is not in a window, it is an **X**. When the mouse pointer is in a window, it changes to an I-beam or an arrow.

Use the mouse and the mouse buttons to make temporary changes to windows and icons. The window system's control files must be altered to make permanent changes to windows and icons. The *Modifying the Default Window Configuration* section in Chapter 2 describes how to make those changes. For more information, see the chapter on X in the *Installation/Administration Manual.*

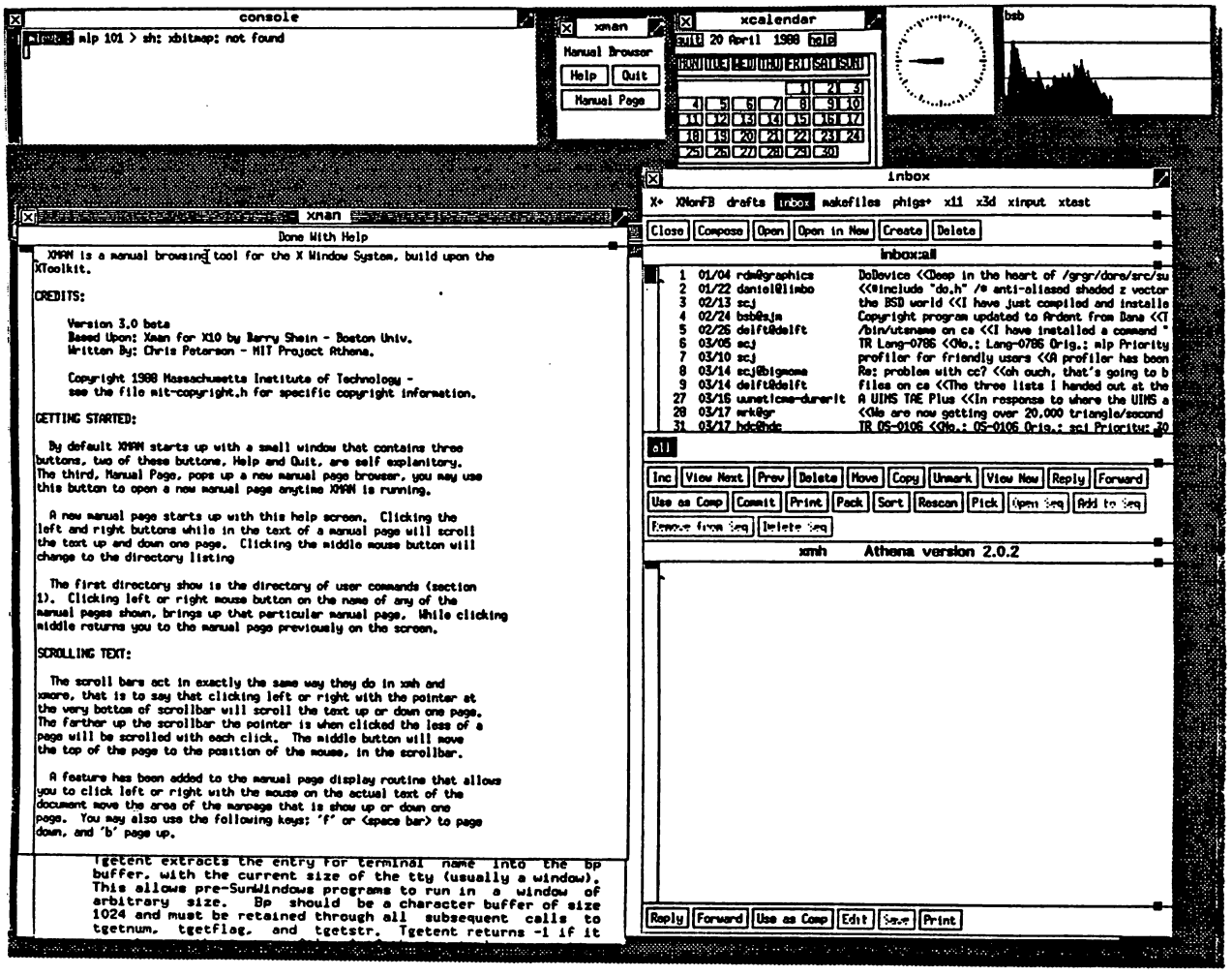


Figure 1-4. Windows and Icons

Simple Commands

When the prompt appears on your screen, the operating system has recognized you as an authorized user and is waiting for you to request a program by entering a command.

Move the mouse around on the mouse pad until the mouse pointer appears in one of the windows. Then, try running the **date** command. Type the command and press ↵. The operating system accesses a program called **date**, executes it, and prints its results on the screen, as shown below.

```
Stardent 1500/3000: date↵
Wed Oct 15 09:49:44 EDT 1986
Stardent 1500/3000:
```

As you can see, the **date** command prints the date and time using the 24-hour clock. Note that the **date** command accepts no options or arguments.

Here are a couple of other commands to try.

who -q ↵

This command gives the names of users currently logged into your system, together with a count of users. The **-q** option means that this is a "quick" or short version of the command. Notice there are no arguments given.

echo Hello There! ↵

This command prints all the words on the line following the command **echo**. "Hello" and "There!" are arguments; there are no options.

Logging Off

When you have completed a session with the operating system, type CTRL d or **exit** after the prompt. (Remember that you type control characters such as CTRL d by holding down the control key and pressing the appropriate alphabetic key. Because control characters are nonprinting characters, they do not appear on the screen.) After several seconds, the *login:* prompt appears on the screen.

```
Stardent 1500/3000: CTRL d
login:
```

This prompt shows that you have logged off successfully and the system is ready for the next login.

DISPLAY

CHAPTER TWO

Contents

- *A. Session One*: Starting up Stardent's implementation of the X Window System.
- *B. Basics*: Looking at X from the ground up.
- *C. Resources*: Where to find more information; note the availability of the on-line *man* feature for X topics.
- *D. Quick Ref*: A list of the most frequently used commands.

Introduction

The X+ Window System is an implementation of the standard X Window System 11 Release 3, developed by the Massachussetts Institute of Technology's Project Athena and the X Consortium, a group of companies responsible for developing and standardizing extensions to X.

Material presented here is for those without X experience. If you are a programmer who wants to use X in an application, check Section B of this chapter and Appendix B at the end of the volume for sources of information.

A. Session One

To start X:

xstart ↵

To get help for X:

manxterm ↵

To call up the WindowOps menu:

Place pointer on root window and hold down the right mouse button.

To manipulate windows:

From the *WindowOps* menu, select *Resize* to resize a window, *MoveOpaque* to move it, *Raise* to raise it, *Lower* to lower it, *Destroy* to delete it, *Iconify* to change a window into an icon or vice versa.

To quit X and return to the UNIX shell prompt:

xexit ↵

B. Basics

When you enter X by typing **xstart** , a screen image is generated with windows and icons. Figure 2-1 is a picture of a screen with the default windows and icons. Think of the screen as a desktop and the windows on the screen as sheets of paper in stacks.

You can run programs, edit files, and send mail messages within each window, just as on an ordinary screen which has only one window — the screen itself.

Icons represent closed windows. You can close windows into icons and open icons into windows. You must open an icon before you can use the window the icon represents.

Your Stardent 1500/3000 is shipped with a default window configuration, as shown in Figure 2-1. This configuration displays the following windows:

left_shell	For interaction with the shell.
right_shell	For interaction with the shell.
console	For system messages such as disk space availability, system errors, and so on.
clock	Visual clock.
load_average	Graph of system activity over time.

You can temporarily or permanently modify the default windows. You can

- Open and close windows.
- Resize windows.
- Move windows.
- Raise and lower windows.

Use the mouse to modify windows temporarily; alter the *.Xdefaults* and *.xdesktop* files in your home directory to modify windows permanently. See *Modifying the Default Window Configuration* section in this chapter for more information on these files. Also, the chapter on X in the *System Administrator's Manual* deals in detail with altering the default configuration.

Figure 2-1. Default Windows

Using the Mouse

The mouse pointer is your tool to point at objects on the screen. Move the mouse around on the mouse pad to move the mouse pointer on the screen.

When the mouse pointer is outside a window (such as in the blank area of the screen called *background*), it appears in the shape of an **X**; when the mouse pointer is moved inside a window, it changes shape to an application-specific cursor.

Figure 2-1 shows the mouse pointer inside the right window. If you type text or issue a command when the mouse pointer is outside a window, nothing happens. You must move the mouse pointer inside a window before you can work in the window.

Do not confuse the mouse pointer with the cursor. The cursor is the small rectangular character that tracks your current type-in location within the window. (See Figure 2-1.) Moving the mouse does not move the cursor. When the mouse pointer is outside a window the cursor is an open rectangle; it becomes solid when the mouse pointer is inside a window.

Each window has a title area at the top of the window that displays the name of the window and contains a close box and a resize box. See Figure 2-1.

- Moving the pointer into the close box and pressing any of the three mouse buttons will iconify the window.
- Moving the pointer into the resize box and pressing any button allows you to resize the window.
- Pressing any button while the pointer is in the title area (but not inside the close or resize boxes) allows you to move the window.

You can also modify windows temporarily by using the three mouse buttons in conjunction with the ALT key and the SHIFT key on your keyboard. Table 2-1 shows the modifications you can make.

The table below applies to the default mouse button bindings defined for Stardent 1500/3000. It is possible to change the default bindings, but it is not recommended until you gain more experience with X because changing the bindings can affect some of the programs. See *Modifying the Default Mouse Keys* later in this chapter.)

Here is how to use combinations of keys and mouse buttons:

SHIFT Click MIDDLE

Hold down the SHIFT key on the keyboard as you press and release the MIDDLE mouse button.

ALT Click LEFT

Hold down the ALT key on the keyboard as you press and release the LEFT mouse button.

ALT Change LEFT

Hold down the ALT key on the keyboard as you press and

Table 2-1. Default Mouse Buttons

Keyboard Key	Context	Action	Mouse Button		
			LEFT	MIDDLE	RIGHT
Shift	Window	Click	Raise	Circulate	Lower
Alt	Window	Click	Raise (open or close)	Iconify	Lower
Alt	Window	Change	Move	Resize	
Alt Shift *or* *(background)*	Window	Hold	User-Menu1	User-Menu2	System-Menu

hold down the LEFT mouse button and move the mouse.

ALT SHIFT Hold RIGHT

Hold down the ALT and SHIFT keys on the keyboard as you press and hold down the RIGHT mouse button and move the mouse. Continue to hold down the mouse button to retain the menu on your screen.

background Hold RIGHT

Move the mouse pointer to the background area of your screen (not in a window) and hold down the ALT button as you press and hold down the RIGHT mouse button. Continue to hold down the mouse button to retain the menu on your screen.

Opening and Closing Windows

To open a window:

(1) Move the mouse pointer into the icon you want to open.

(2) ALT Click MIDDLE.

The icon opens into a window and positions itself on the screen as specified in your *.xdesktop* file.

To close a window:

(1) Move the mouse pointer into the window you want to close.

(2) ALT Click MIDDLE.

The window closes and is displayed as an icon.

Windows remain open until you close them; icons remain closed until you open them. When you log out and log back in, the windows appear as configured in the *.xdesktop* file.

Resizing Windows

To resize a window:

(1) Move the mouse pointer into the window you want to resize.

(2) ALT Change MIDDLE. A grid and a shaded size indicator appear in the window. The size indicator adjusts as you move the mouse up, down, left, right, or diagonally. The size indicator shows the size of the resized window. For terminal emulator (*xterm*) windows, it shows the number of rows and columns; for graphical windows (such as the clock) the size is given in pixels.

The extent to which you can resize a window depends on the location of the mouse pointer when you do

ALT Change MIDDLE

Specifically, if the mouse pointer is near an edge of a window, only that side will be moved. If the mouse pointer is near a corner, both sides can be moved.

The window stays resized until you resize it again or until you logout. When you logout and log back in, the window returns to the size set in the *.xdesktop* file.

The best way to learn how to resize windows is to experiment with the mouse pointer.

Moving Windows

To move a window:

(1) Position the mouse pointer in the window you want to move.

(2) ALT Change LEFT. A grid appears and moves with the window as you move the mouse up, down, left, right, or diagonally.

Raising, Lowering, and Circulating Windows

If you open windows on top of other windows, you need to know how to move the bottom window to the top (raise the window) and move the top window to the bottom (lower or push the window). If you have several windows stacked on top of each other you can also *circulate* the windows (move the windows up step-by-step in the stack). To raise a window:

(1) Move the mouse pointer into the window you want to raise.

(2) SHIFT Click LEFT

or

ALT Click LEFT.

To lower a window:

(1) Move the mouse pointer into the window you want to lower.

(2) SHIFT Click RIGHT

or

ALT Click RIGHT.

To circulate windows:

(1) Move the mouse pointer into top window in the stack you want to circulate.

(2) SHIFT Click MIDDLE.

The System Menu

To bring up the system menu, use ALT SHIFT Hold RIGHT or *background* Hold RIGHT, is shown in Table 2-2.

Table 2-3 shows the functions represented by the menu.

Table 2-2. System Menu

RefreshScreen
Resize
Lower
Raise
Preferences
CircUp
CircDown
MoveOpaque
Iconify
Focus
Destroy
Restart
Exit X Windows

Table 2-3. System Menu Functions

Name	Function
RefreshScreen	Refreshes (redraws) the entire display
Resize	Resize a window
Lower	Lower a window
Raise	Raise a window
Preferences	Display the Preferences menu
CircUp	Causes a window to circulate up
CircDown	Causes a window to circulate down
MoveOpaque	Move a window opaquely
Iconify	Iconify a window (deiconify an icon)
Focus	Focus the keyboard on a window
Destroy	Destroy a window
Restart	Reinitialize the window manager
Exit X Windows	Exit from the window system

Moving the mouse pointer to one those entries that appear with a right arrow and sliding the pointer out of the window to the right brings up a menu for that entry. For instance, moving the mouse pointer to the Preferences entry and sliding the pointer out of the window to the right will display the User Preferences menu.

The User Preferences menu makes it possible to control the window manager's behavior. Table 2-4 gives the choices available in the User Preferences menu.

Table 2-4. User Preferences Menu

Name	Function
Bell Loud Bell Normal Bell Off	Change the volume of the bell
Click Loud Click Soft Click Off	Change the key click volume
Mouse Fast Mouse Normal Mouse Slow	Change Mouse speed
Screensaver on Screensaver off	Enable/Disable the screen saver
Highlight Don't Highlight	Do/Don't Highlight the window titles when the mouse is in the window
Autoraise Don't Autoraise	Do/Don't Auto Raise windows when the mouse moves into it
Rootbox	Place the resize box in the upper left corner of the RootWindow rather than in the resized window. This prevents potentially annoying double exposures when set.

The User Menu

The default Applications menu (shown in Table 2-1 as User-Menu1), which you bring up with ALT SHIFT Hold LEFT or *background* Hold LEFT, has the options shown in Table 2-5.

Make selections by moving the mouse pointer to your choice of function and releasing the mouse button.

Table 2-5. Options for Default Applications Menu

Name Function	
xterm	Runs a shell
xcalc	A simple programmable calculator
xclock	An analog clock
xlock	A screen lock facility
xman	A man page bowser
xload	Displays the systems load average

Modifying the Default Window Configuration

The *.xdesktop* and *.Xdefaults* files in your home directory control the default configuration of your windows. Every time you log in, windows display on your screen as specified in these files. If you want to change your default window configuration permanently, you need to modify one or both of these files.

The *.xdesktop* file in your home directory specifies the initial set of windows, their layout and appearance. It is actually a shell script executed by the window manager when it is started up. If you want to change the layout of your desktop you need to change this file. The default *.xdesktop* file looks like this:

```
#!/bin/sh
xrdb -merge $HOME/.Xdefaults
xset s 360 m 4 2
xclock -analog -g 100x100+830+0&
xload -g 200x100+940+0&
xterm -g 80x8+0+0 -C -n console&
xterm -g 80x50+0+200 -n `hostname`&
xterm -g 80x50+550+200 -n `hostname`&
```

The *.Xdefaults* file in your home directory is used to specify default values for such items as fonts, colors, window sizes/layouts, cursors, and so on. It contains lines of the form

program.option: value

that allow you to define values for a wide variety of application characteristics.

The *.Xdefaults* file should contain the default application characteristics you most often want. If you want some special value for some application, override *.Xdefaults* values by specifying the special value on a command line.

Modifying the Default Mouse Keys

The system *.awmrc* file controls the default bindings of your mouse buttons. If you want to change default mouse button defaults permanently, modify the *.awmrc* file.

Changing the mouse button bindings can adversely affect some of the X programs. Before attempting to redefine the mouse pointer buttons see the *awm* Client *man* pages in the *Window System Manual.*

C. Resources

The on-screen *man* help facility has information about all aspects of X. Use the *xman* command that provides you with an interactive tool for browsing through X *man* pages.

Ardent's *Window System Manual* contains printouts of X user *man* pages and chapters on server and stereo library extensions, as well as on multiple buffering.

The Ardent *Window System Toolkit* reference manual has printouts of *man* pages for *Xt* and *Xw* widgets, plus MIT documentation on *X Toolkit Athena Widgets C Language Interface.*

The Ardent *System Administrator's Manual* has a fairly substantial chapter on modifying X defaults.

General Literature

Scheifler, Gettys, Newman: *X Window System C Library and Protocol Reference.* Digital Press, 1988.

Oliver Jones: *Introduction to the X Window System.* Prentice-Hall, 1989.

Operation	Description
Enter X Window	Type **xstart**
Open Window Close Window	Move mouse pointer to icon/window; hold down ALT key as you press and release middle mouse button.
Resize Window	Move mouse pointer to window; hold down ALT key as you press and hold down middle mouse button. Move mouse and observe size indicator.
Move window	Move mouse pointer to window; hold down ALT key as you press and hold down left mouse button. Move mouse and observe grid.
Raise window	Move mouse pointer to window; hold down ALT key as you press and release the left mouse button.
Lower window	Move mouse pointer to window; hold down ALT key as you press and release the right mouse button.
Circulate windows	Move mouse pointer into top window; hold down the shift key as you press and release the middle mouse button.
Call up System Menu	Place pointer on root window and hold down the right mouse button.
Delete window	Select **Destroy** from the System Menu.
Redraw screen	Select **RefreshScreen** from the System Menu.
Restart X Window	Select **Restart** from the System Menu.
Help	Type **man xterm**.
Exit X Window	Type **xexit**.

UNIX ESSENTIALS

CHAPTER THREE

Contents

- *A. Session One*: Jump in and start using the operating system.
- *B. Basics*: Bare-bone essentials about UNIX; B-1 is about the file system, B-2 about the shell.
- *C. Resources*: Where to find more information; note the availability of the on-line *man* feature for UNIX topics.
- *D. Quick Ref*: A list of the most frequently used commands.

Introduction

This chapter is about the UNIX operating system, dealing specifically with the file system and the "shell" — the command interpreter that provides the interface between you and Stardent 1500/3000.

After *Session One*, you will find sections on *How the File System is Structured* and *Your Place in the File System*, followed by the introduction of commands that enable you to build your own directory structure, access and manipulate the subdirectories and files you organize within it, and examine the contents of other directories in the system.

After examining basics of the file system, we'll look at the C-shell, an enhanced version of the UNIX command interpreter, which can be used both as a command interface and programming language.

The section on the shell introduces you to commands that enable you to find files with pattern matching, run commands in the background, schedule when commands are to be executed or run groups of commands sequentially, redirect standard input and output from and to files, and other commands.

A. Session One

To make a new directory:

mkdir *mydir* ↵

To move into the directory (change directory):

cd *mydir* ↵

To move back into your home directory:

cd ↵

To see what's in the directory:

ls -l ↵

To print a file on the screen (concatenate):

cat *you-named-it* ↵

To find out where you are (print working directory):

pwd ↵

B-1. Basics: The File System

To use the file system effectively, you need to know its structure, described below. Next, you will find some basic file system commands, which you can best learn by trying them as you read about them.

How the File System is Structured

The file system is a set of ordinary files, special files, and directories. It provides a way to organize, retrieve, and manage information electronically. Briefly,

- An ordinary file is a collection of characters stored on a disk. It may contain text for a report or code for a program.
- A special file represents a physical device, such as a terminal or disk.
- A directory is a collection of files and other directories (sometimes called subdirectories). You can use directories to group files together on the basis of any criteria you choose.

For example, you might create a directory for each product that your company sells or for each of your student's records.

The set of all the directories and files is organized into a tree-shaped structure. Figure 3-1 shows a sample file structure with a directory named *root* (/) as its source. By moving down the branches extending from root, you can reach other major system directories. By branching down from these, you can, in turn, reach all the directories and files in the file system.

In this hierarchy, files and directories subordinate to a directory have a parent-child relationship. This type of relationship is possible for many layers of files and directories. In fact, there is no limit to the number of files and directories you may create in any directory that you own. Neither is there a limit to the number of layers of directories that you may create, so you have the capability to organize files in a variety of ways.

Your Place in the File System

Whenever you interact with the operating system, you do so from a location in its file system structure. The operating system automatically places you at a specific point in its file system every time you log in. From that point you can move through the hierarchy to work in any of your directories and files and to access those belonging to others that you have permission to use.

The following paragraphs describe your position in the file system structure and how this position changes as you move through the file system.

Your Home Directory

When you successfully complete the login procedure, the operating system places you at a specific point in its file system structure called your login or home directory. Each user assigns a name to his or her home directory. Most users give their home directory the same name as their login name. If you share a Stardent 1500/3000 with other users, each user has a personal home directory.

Within your home directory you can create files and additional directories in which to group them. You can move and delete these files and directories, and also control access to them. Your home directory is a jumping-off point from which to view all the

files of the file system, all the way up to *root*.

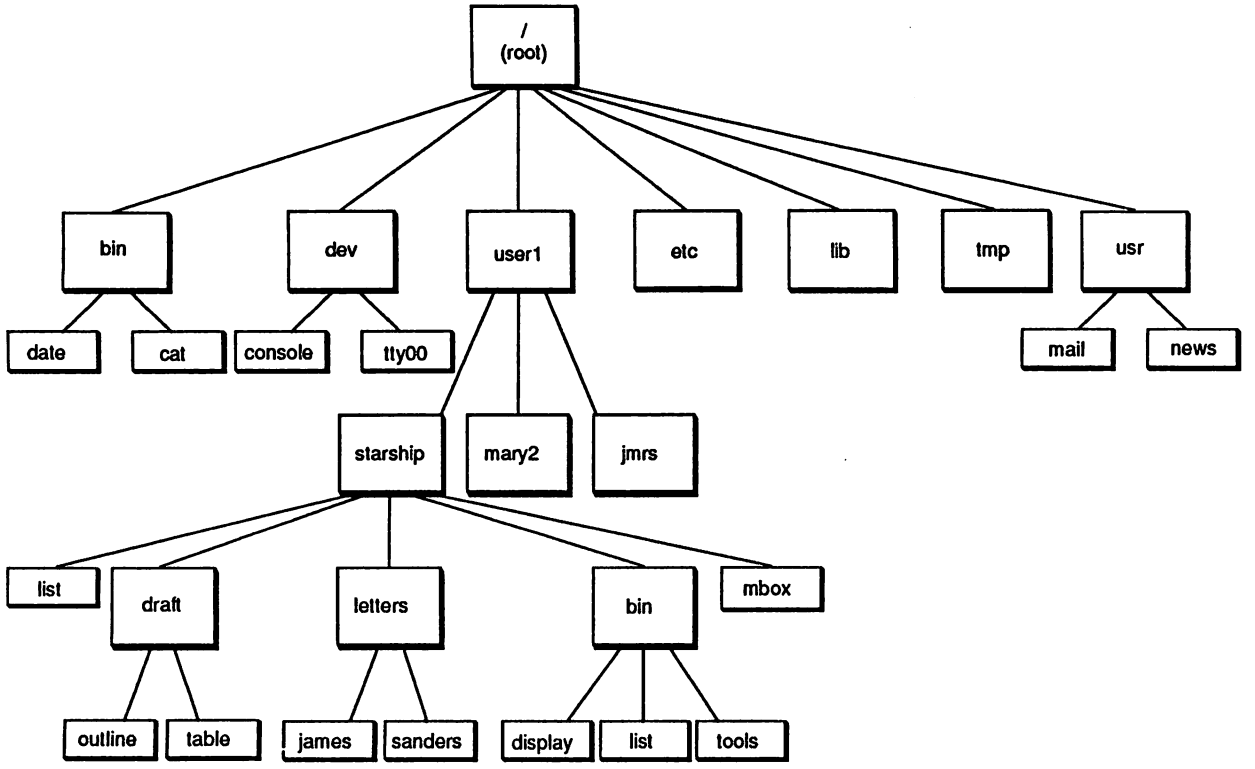


Figure 3-1. Sample File System

Your Current Directory

As long as you continue to work in your home directory, it is considered the current working directory. If you move to another directory, that directory becomes the current working directory.

The **pwd** command (short for print working directory) prints the name of the directory in which you are now working. For example, if your login name is *starship* and you execute the **pwd** command in response to the first prompt after logging in, the system responds as follows:

```
Stardent 1500/3000: pwd↵
/user1/starship
Stardent 1500/3000:
```

The system response gives you both the name of the directory in which you are working (*starship*) and the location of that directory in the file system. The name */user1/starship* tells you that the *root* directory (shown by the leading / in the line) contains the directory *user1* which in turn contains the directory *starship*. (All other slashes in the pathname other than / are used to separate the names of directories and files and to show the position of each directory relative to /. A directory name that shows the directory's location in this way is usually called a full pathname.

Remember, you can determine your position in the file system at any time simply by issuing the **pwd** command. This is especially helpful if you want to read or copy a file and the operating system tells you the file does not exist. You may be surprised to find you are in a wrong directory.

Pathnames

Every file and directory in the system is identified by a unique pathname. The pathname shows the location of the file or directory, and provides directions for reaching it. Knowing how to follow the directions given by a pathname is your key to moving around the file system successfully. The first step in learning about these directions is to learn about the two types of pathnames: full and relative.

Full Pathnames

A full pathname (sometimes called an absolute pathname) gives directions that start in the root directory and lead you down through a unique sequence of directories to a particular directory or file. You can use a full pathname to reach any file or directory.

Because a full pathname always starts at the root of the file system, its leading character is always a / (slash). The final name in a full pathname can be either a file name or a directory name. All other names in the path must be directories.

To understand how a full pathname is constructed and how it directs you, consider the following example. Suppose you are working in the *starship* directory, located in */user1*. You issue the **pwd** command and the system responds by printing the full pathname of the current working directory: */user1/starship*. Analyze the elements of this pathname using the following diagram and key.

/ (leading)	The slash that appears as the first character in the pathname is the *root* of the file system.
user1	The subdirectory one level below *root* in the hierarchy to which *root* points or branches.
/ (subsequent)	The separator between the *user1* directory name and the *starship* directory name.
starship	The current working directory.

Figure 3-2 is a diagram of the full pathname of the *starship* directory. Follow the bold lines in the figure to trace the full path to */user1/starship*.

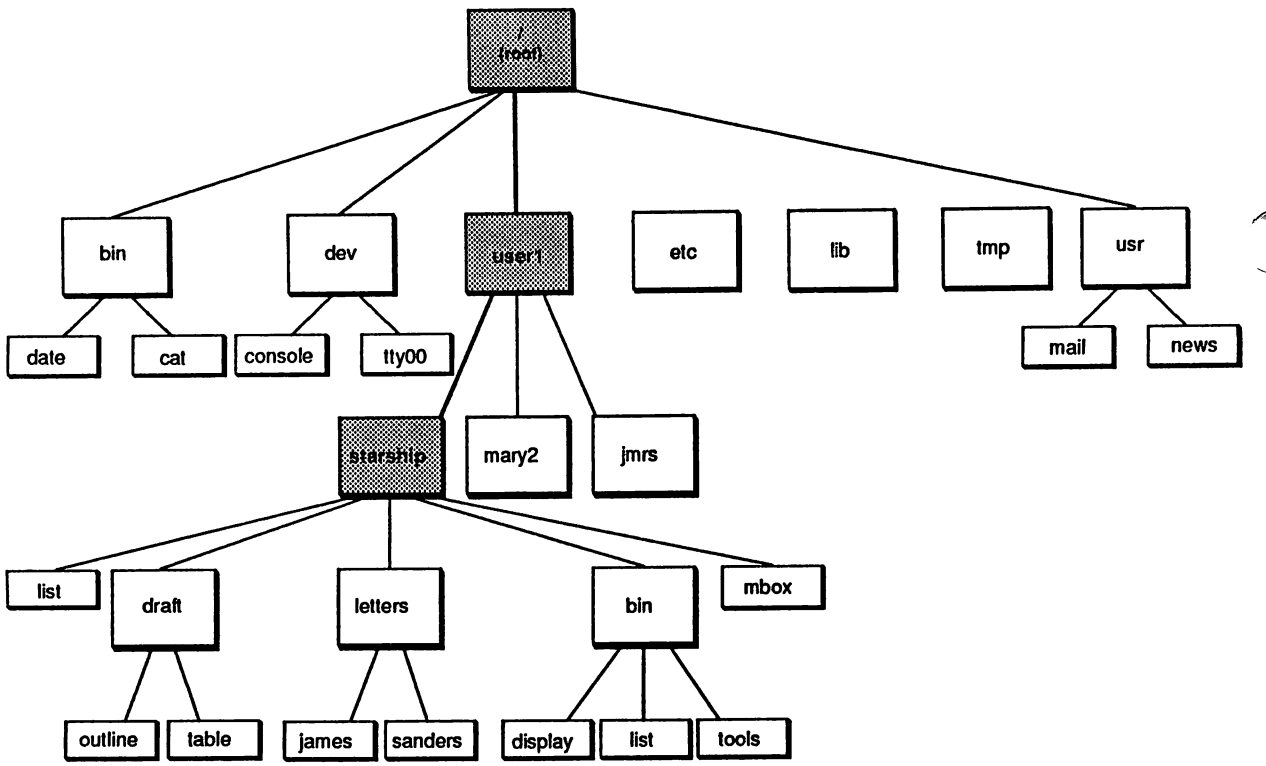


Figure 3-2. Full Path for the *starship* Directory

Relative Pathnames

A relative pathname gives directions that start in the current working directory and lead you up or down through a series of directories to a particular file or directory. By moving down from the current directory you can access files and directories you own. By moving up from the current directory you pass through layers of parent directories to the parent of all / directories. From there you can move anywhere in the file system.

A relative pathname begins with one of the following:

- A directory or file name.
- A . (pronounced dot), which is a shorthand notation for the current directory.
- A .. (pronounced dot dot), which is a shorthand notation for the directory immediately above the current directory in the file system hierarchy.

The directory represented by .. (dot dot) is called the parent directory of . (the current directory).

For example, say you are in the directory *starship* in the sample system and *starship* contains directories named *draft*, *letters*, and *bin* and a file named *mbox*. The relative pathname to any of these is simply its name, such as *draft* or *mbox*. Figure 3-3 traces the relative path from *starship* to *draft*.

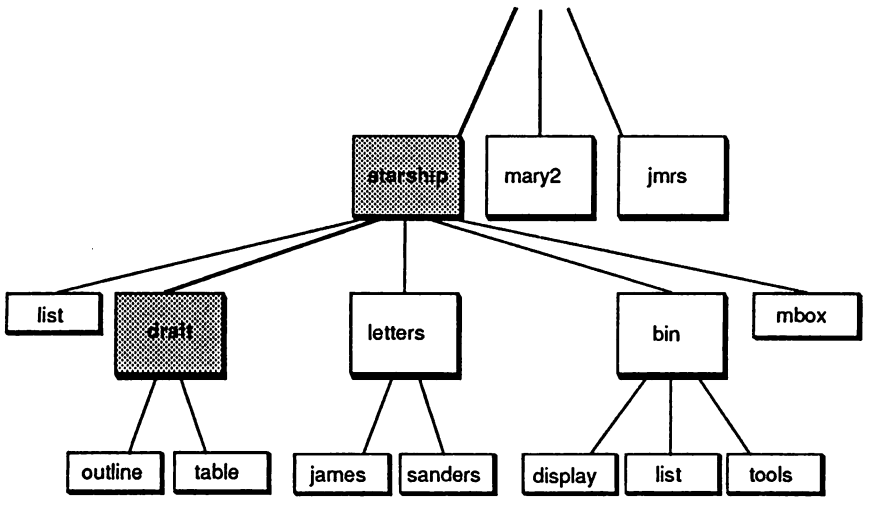


Figure 3-3. Relative Path for the *draft* Directory

The *draft* directory belonging to *starship* contains the files *outline* and *table*. The relative pathname from *starship* to the file *outline* is

draft/outline. Figure 3-4 traces this relative path. Notice that the slash in this pathname separates the directory named *draft* from the file named *outline*. Here, the slash is a delimiter showing that *outline* is subordinate to *draft*; that is, *outline* is a child of its parent, *draft*.

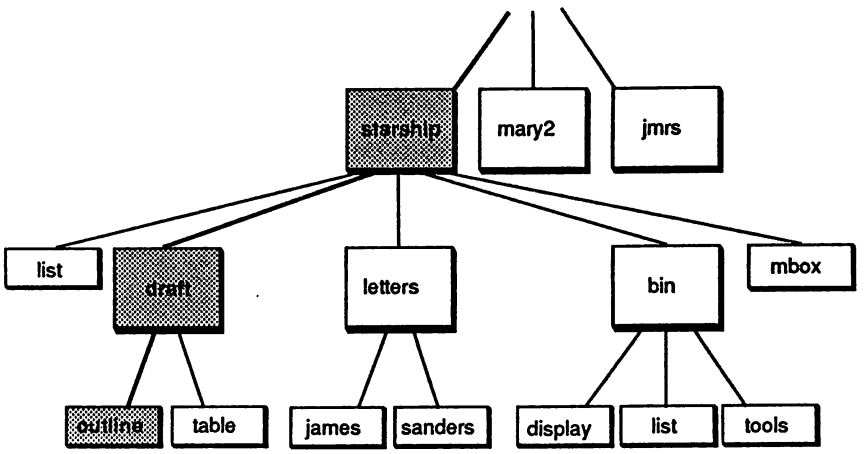


Figure 3-4. Relative Path from *starship* to *outline*

So far, the discussion of relative path names has covered how to specify names of files and directories that belong to, or are children of, the current directory. You now know how to move down the system hierarchy level by level until you reach your destination. You can also, however, ascend the levels in the system structure or ascend and subsequently descend into other files and directories.

To ascend to the parent of the current directory use the .. notation. This means that if you are in the directory named *draft* in the sample file system, .. is the pathname to *starship*, and *../..* is the pathname to *starship*'s parent directory, *user1*.

From *draft* you can also trace a path to the file *sanders* by using the pathname *../letters/sanders*. The .. brings you up to *starship*. Then the names *letters* and *sanders* take you down through the *letters* directory to the *sanders* file.

Keep in mind that you can always use a full pathname in place of a relative one. Some examples of full and relative pathnames are:

/ Full pathname of the root directory.

/bin Full pathname of the *bin* directory (contains most executable programs and utilities)

/user1/starship/bin/tools
Full pathname of the *tools* directory belonging to the *bin* directory that belongs to the *starship* directory belonging to *user1* that belongs to *root*.

bin/tools Relative pathname to the file or directory *tools* in the directory *bin*.

If the current directory is / the operating system looks for */bin/tools*. If the current directory is *starship* the system looks for the full path */user1/starship/bin/tools*.

tools Relative pathname of a file or directory *tools* in the current directory.

Naming Directories and Files

You can give directories and files any names you want as long as the name conforms to the following rules:

- The name of a directory (or file) can be from one to fourteen characters long.
- All characters other than / are legal.
- It is best to avoid using a space, tab, backspace, and the following:

  ? @ # $ ^ & * ( ) ` [ ] \ | ; ' " < >

  If you use a blank or tab in a directory or file name, you must enclose the name in quotation marks on the command line.
- Do not use a +, – or . as the first character in a file name. They have special meanings.
- Upper case and lower case characters are distinct to the operating system. For example, the system considers a directory (or file) named *draft* to be different from one named *DRAFT*.

The following are examples of legal directory or file names:

memo	*MEMO*	*section2*	*ref:list*
file.d	*chap3+4*	*item1-10*	*Outline*

Organizing a Directory

This section introduces four commands that allow you to organize and use a directory structure: **mkdir, ls, cd,** and **rmdir.**

mkdir Makes new directories and subdirectories.

ls Lists the names of all the subdirectories and files in a directory.

cd Changes your location in the file system from one directory to another.

rmdir Removes an empty directory.

These commands can be used with either full or relative pathnames. Two of the commands, **ls** and **cd,** can also be used without a pathname. Each command is described more fully in the four sections that follow.

Creating Directories: the mkdir *Command*

It is a good idea to create subdirectories in your home directory according to a logical and meaningful scheme that facilitates the retrieval of information from your files. If you put all files pertaining to one subject together in a directory, you know where to find them later.

To create a directory, use the **mkdir** (make directory) command, followed by the name you are giving the new directory. For example, in the sample file system, the owner of the *draft* subdirectory created *draft* by issuing the following command from the home directory (*/user1/starship*):

```
Stardent 1500/3000: mkdir draft↵
Stardent 1500/3000:
```

The second prompt shows that the command has succeeded; the subdirectory *draft* has been created.

The **mkdir** command allows you to create several directories at once. For instance, the following command line creates the directories *draft, letters,* and *bin.*

```
Stardent 1500/3000:  mkdir draft letters bin↵
Stardent 1500/3000:
```

You can also move to a subdirectory you created and build additional subdirectories within it. You can name subdirectories and files anything you want as long as you follow the guidelines listed earlier in this chapter under *Naming Directories and Files.*

Listing the Contents of a Directory: the ls *Command*

All directories in the file system maintain information about the files and directories they contain, such as name, size, and the date last modified. You can obtain this information about the contents of your current directory and other system directories by executing the **ls** (short for list) command.

The **ls** command lists the names of all files and subdirectories in a specified directory. If you do not specify a directory **ls** lists the names of files and directories in the current directory. To understand how the **ls** command works consider the sample file system shown in Figure 3-1.

Say you are logged in and run the **pwd** command. The system responds with the pathname */user1/starship.* To display the names of files and directories in this current directory you then type

ls ↵

After this sequence your screen reads:

```
Stardent 1500/3000:  pwd↵
/user1/starship
Stardent 1500/3000:  ls↵
Memos
bin
draft
letters
list_1
list_two
mbox
Stardent 1500/3000:
```

As you can see, the system responds by listing the names of files and directories in the current directory *starship*. Files are listed in ASCII order, with numbers and upper case letters printed before lower case.

To print the names of files and subdirectories in another directory without moving from the current directory, you must specify the name of that directory as follows:

ls *pathname* ↵

The directory name can be either the full or relative pathname of the desired directory (and use the ..(dot dot) notation if you wish). To list the contents of *draft* while you are working in *starship*, type

```
Stardent 1500/3000: ls draft↵
outline
table
Stardent 1500/3000:
```

Here, *draft* is a relative pathname from a parent (*starship*) to a child (*draft*) directory.

You can also use a relative pathname to print the contents of a parent directory when you are located in a child directory. For example, the following command line uses the .. (dot dot) notation to specify the relative pathname from *starship* to *user1*:

```
Stardent 1500/3000: ls ..↵
jmrs
mary2
starship
Stardent 1500/3000:
```

You can get the same results by using the full pathname from / to *user1*. If you type

ls /user1 ↵

the system responds by printing the same list.

Similarly, you can list the contents of any directory for which you have access by executing the **ls** command with a full or relative pathname.

The **ls** command is useful for long lists of files when you want to find out whether one of them exists in the current directory. For example, if you are in the directory *draft* and want to determine if the files named *outline* and *notes* are there, use the **ls** command as follows:

```
Stardent 1500/3000: ls outline notes↵
outline
notes not found
Stardent 1500/3000:
```

The system acknowledges the existence of *outline* by printing its name and says that *notes* is not found.

The **ls** command does not print the contents of a file. If you want to see what a file contains use the **cat**, **pg**, or **pr** commands described in *Accessing and Manipulating Files*, later in this chapter.

The **ls** command also accepts various options. Three of these, the **–a**, **–l**, and **–F** options, give additional information not included in the basic **ls** command, while the **–C** option yields a columnar file list. (See *How to Execute Commands* in Chapter 1 for a general discussion about using options in command lines. For additional options see the listing **ls**(1) in the *Commands Reference Manual.*)

Listing All Names in a File: the –a Option

Some important file names in your home directory, such as **.profile** (pronounced dot-profile), begin with a dot. When a file name begins with a dot it is not included in the list of files reported by the **ls** command. If you want the **ls** to include these files use the **–a** option on the command line.

For example, to list all the files in your current directory (*starship*), including those that begin with a . (dot), type

```
Stardent 1500/3000: ls -a↵
.
..
.profile
Memos
bin
draft
letters
list_1
list_two
```

```
mbox
Stardent 1500/3000:
```

Listing Contents in Short Format: the –C and –F Options

NOTE
In C-shell the ls automatically lists files in columns. The –C option is unnecessary.

The –C and –F options for the ls command are particularly useful when listing the contents of large directories. The –C option lists a directory's subdirectories and files in columns, while the –F option identifies executable files (with an *) and directories (with a /). For example, to list the files in your working directory *starship* use the command line shown here:

```
Stardent 1500/3000: ls -CF↵
bin/    letters/        mbox
draft/  list*
Stardent 1500/3000:
```

Listing Contents in Long Format: the –l Option. The most informative ls option is –l, which displays the contents of a directory in long format, giving mode, number of links, owner, group, size in bytes, and time of last modification for each file. For example, say you run the ls –l command while in the *starship* directory.

```
Stardent 1500/3000: ls -l↵
total 30
drwxr-xr-x   3 starship      project        96  Oct 27  08:16  bin
drwxr-xr-x   2 starship      project        64  Nov  1  14:19  draft
drwxr-xr-x   2 starship      project        80  Nov  8  08:41  letters
-rwx------   2 starship      project     12301  Nov  2  10:15  list
-rw-------   1 starship      project        40  Oct 27  10:00  mbox
Stardent 1500/3000:
```

The first line of output

```
total 30
```

shows the amount of disk space used, measured in blocks. (A block is 512 bytes, or one-half a kilobyte.) Each of the other lines contains a report on a directory or file in *starship*. The first character in each line (`d`, `-`, `b`, or `c`) tells you the type of file.

NOTE
A byte is the smallest addressable unit of memory (8 bits). Each character is stored in a byte, so the number of bytes in an ls listing equals the number of characters in the file or directory.

`d` Indicates a directory.

`-` Indicates an ordinary disk file.

b Indicates a block special file.

c Indicates a character special file.

Using this key to interpret the previous screen, you can see that the *starship* directory contains three directories and two ordinary disk files.

The next several characters, which are either letters or hyphens, identify who has permission to read and use the file or directory. Permissions are discussed in the description of the **chmod** command under *Accessing and Manipulating Files* later in this chapter.

The following number is the link count. For a file this equals the number of parent and grandparent directories the file has, and for a directory it includes in addition the number of directories immediately under it in the file structure.

Next, the login name of the file's owner appears (here it is *starship*), followed by the group name of the owner of the file or directory (*project*).

The following number shows the length of the file or directory entry measured in bytes (see the note above). (For a directory this includes just the byte count for the directory itself, not for its files or sub-directories.) The month, day, and time that the file was last modified is given next. Finally, the last column shows the name of the directory or file.

Figure 3-5 identifies each column in the rows of output from the **ls -l** command.

Changing the Current Directory: the cd *Command*

When you first log in you are placed in your home directory. It is, for the time being, also the current working directory. By using the **cd** (short for change directory) command, you can work in other directories as well. To use this command enter **cd** followed by a pathname to the directory to which you want to move.

cd *pathname_of_newdirectory* ↵

Any valid pathname (full or relative) can be used as an argument to the **cd** command. If you do not specify a pathname the command moves you to your home directory. Once you have moved

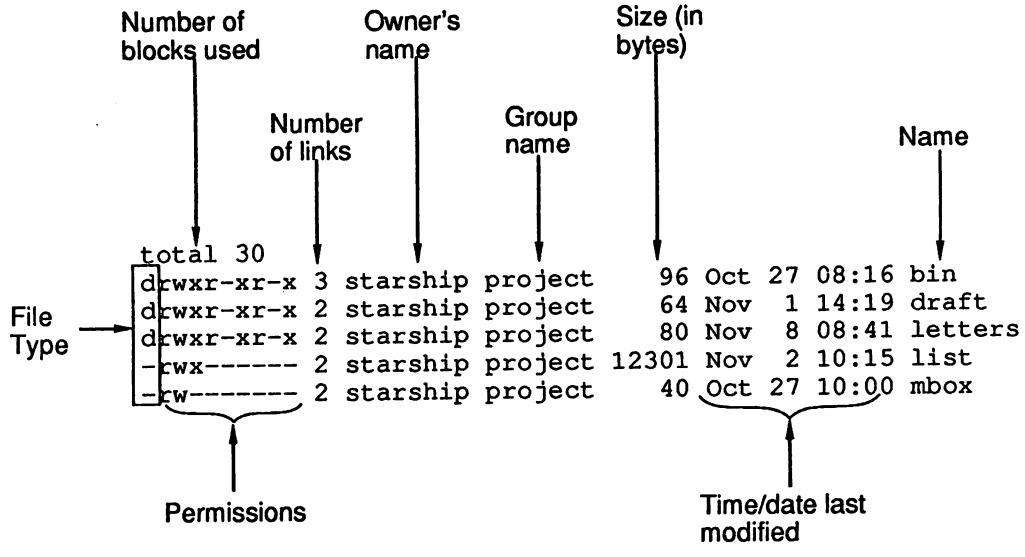


Figure 3-5. Output of the ls -1 command

to a new directory it becomes the current working directory.

For example, to move from the *starship* directory to its child directory *draft* (in the sample file system), type

cd draft ↵

(Here *draft* is the relative pathname to the desired directory.) When you get a prompt, verify your new location by typing

pwd ↵

Your terminal screen looks like this:

```
Stardent 1500/3000: cd draft↵
Stardent 1500/3000:  pwd↵
/user1/starship/draft
Stardent 1500/3000:
```

Now that you are in the *draft* directory you can create subdirectories in it by using the **mkdir** command, and new files by using the **vi** or **ed** editors, described later in this manual.

You can also use full pathnames with the **cd** command. For example, to move to the *letters* directory from the *draft* directory, specify */user1/starship/letters* on the command line, as follows:

cd /user1/starship/letters ↵

Also, because *letters* and *draft* are both children of *starship*, use the relative pathname **../letters** with the **cd** command. The .. notation moves you to the directory *starship*, and the rest of the pathname moves you to *letters*.

Removing Directories: the **rmdir** *Command*

If you no longer need a directory, you can remove it with the **rmdir** (short for remove a directory) command:

rmdir *directoryname(s)* ↵

You can specify more than one directory name on the command line.

For the **rmdir** command to work the directory must be empty, you must be the owner, and you must be in the parent directory of the directory you wish to remove.

If you try to remove a directory that still contains subdirectories and files (that is, is not empty), the **rmdir** command prints the message

directoryname not empty

You must remove all subdirectories and files; only then does the command succeed.

For example, say you have a directory called *memos* that contains one subdirectory, *tech*, and two files, *june.30* and *july.31*. If you try to remove the directory *memos* (by issuing the **rmdir** command from your home directory), the system responds as follows:

```
Stardent 1500/3000: rmdir memos ↵
rmdir:  memos not empty
Stardent 1500/3000:
```

To remove the directory *memos* you must first remove its contents: the subdirectory *tech*, and the files *june.30* and *july.31*. If the *tech* subdirectory is empty you can remove it by executing the **rmdir**

command. For instructions on removing files see *Accessing and Manipulating Files* later in this chapter.

Once you have removed the contents of the *memos* directory, *memos* itself can be removed. First, however, you must move to its parent directory (your home directory). The **rmdir** command does not work if you are still in the directory you want to remove. From your home directory type

rmdir memos ↵

If *memos* is empty the system removes it and returns a prompt.

Accessing and Manipulating Files

This section introduces several commands that access and manipulate files.

cat	Prints the contents of a file on your screen.
pg	Prints the contents of a file on your screen in chunks or pages.
pr	Prints a partially formatted version of a file on your screen.
lp	Requests a paper copy of a file from a printer.
cp	Makes a duplicate copy of an existing file.
mv	Moves or renames a file.
rm	Removes a file.
wc	
chmod	Changes permission modes for a file (or a directory).
diff	Displays differences between two files and changes needed to make them the same.
grep	Searches a file for a specific pattern.
sort	Allows you to sort and merge contents of files.

Concatenate and Print Contents of a File: the cat *Command*

The **cat** (short for concatenate) command outputs the contents of file(s). This output is displayed on the screen unless you tell **cat** to direct it to another file or a new command. For example, say you are located in the directory *letters* (in the sample file system) and you want to display the contents of the file *johnson*. Type the command line shown on the example and you receive the following output:

```
Stardent 1500/3000: cat johnson↵
March 5, 1986

Mr. Ron Johnson
Layton Printing
52 Hudson Street
New York, N.Y.

Dear Mr. Johnson:

I enjoyed speaking with you this morning
about your company's plans to automate
your business.
Enclosed please find
the material you requested
about AB&C's line of computers
and office automation software.

If I can be of further assistance to you,
please don't hesitate to call.

Yours truly,

John Howe
Stardent 1500/3000:
```

To display the contents of two or more files simply type the names of the files you want to see on the command line:

```
Stardent 1500/3000: cat johnson sanders↵
March 5, 1986

Mr. Ron Johnson
Layton Printing
52 Hudson Street
New York, N.Y.

Dear Mr. Johnson:

I enjoyed speaking with you this morning
```

```
.
.

Yours truly,

John Howe

March 5, 1986

Mrs. D.L. Sanders
Sanders Research, Inc.
43 Nassau Street
Princeton, N.J.

Dear Mrs. Sanders:

My colleagues and I have been following, with great interest,
.
.

Sincerely,

John Howe
Stardent 1500/3000:
```

Paging Through the Contents of a File: the pg *Command*

The command **pg** (page) allows you to examine the contents of a file or files page by page on a terminal. The **pg** command displays the text of a file in pages (or chunks) followed by a colon prompt(:), a signal that the program is waiting for your instructions. Available instructions include requests for the command to continue displaying the file's contents a page at a time, and a request that the command search through the file(s) to locate a specific character pattern. A list of instructions accepted by the **pg** command follows.

h	Help; display list of available instructions.
q or **Q**	Quit **pg** perusal mode.
↵	Display next page of text.
l	Display next line of text.
d	Display additional half page of text.
CTRL d	Display additional half page of text.

.	Redisplay current page of text.
CTRL l	Redisplay current page of text.
f	Skip next page of text and display following page.
n	Begin displaying next file you specified on command line.
p	Display previous file specified on command line.
$	Display last page of text in file currently displayed.
/pattern	Search forward in file for specified character pattern.
?pattern	Search backward in file for specified character pattern.

You can type most instructions with a preceeding number. For example:

+1↵	Displays the next page of text..
–1↵	Displays the previous page of text.
1 ↵	Displays the first page of text.

See the *Commands Reference Manual* for a detailed explanation of all available **pg** instructions.

The **pg** command is useful when you want to read a long file or a series of files because the program pauses after displaying each page, allowing time to examine it. The size of the page displayed depends on the terminal. For example, on a terminal capable of displaying twenty-four lines, one page is defined as twenty-three lines plus a line containing a colon. If the file is shorter there are correspondingly fewer lines in the page.

To peruse the contents of a file with **pg**, use the following command line:

pg *filename(s)* ↵

The first page of the file appears on the screen. If the file has more lines in it than can be displayed on one page, a colon appears at

the bottom of the screen. This is a reminder to you that there is more of the file to be seen. When you are ready to read more, press ↵ and **pg** prints the next page of the file. Here is an example.

```
Stardent 1500/3000: pg outline↵
After you analyze the subject for your
report, you must consider organizing and
arranging the material you want to use in
writing it.
      .
      .
      .
An outline is an effective method of
organizing the material.  The outline
is a type of blueprint or skeleton,
a framework for you the builder-writer
of the report; in a sense it is a recipe
: ↵
```

When you press ↵, **pg** resumes printing the file's contents on the screen.

```
that contains the names of the
ingredients and the order in which
to use them.
     .
     .
     .
Your outline need not be elaborate or
overly detailed; it is simply a guide you
may consult as you write, to be varied,
if need be, when additional important
ideas are suggested in the actual writing.
(EOF):
```

Notice the line at the bottom of the screen containing the string (EOF):. This string indicates you have reached the end of the file. The colon prompt is a cue for you to issue another instruction. If you are finished with the **pg** command, type ↵ and the system prompt will appear. If you are not at EOF use **Q** or **q** to receive the system prompt.

The **pg** command can also be used along with another command and a pipe symbol () to allow you to view output of a program page by page. See *Redirecting Output to a Command.*

Print Partially Formatted Contents of a File: the **pr** Command

The **pr** command is used to prepare files for printing. It supplies titles and headings, paginates, and prints a file, in any of various page lengths and widths, on your screen.

If you choose not to specify any of the available options, the **pr** command produces output in a single column that contains sixty-six lines per page and is preceded by a short heading. The five-line heading includes the date and time, file name, and page number.

The **pr** command can be used together with the **lp** command to provide a partially formatted paper copy of text. (See *Requesting a Paper Copy of a File* later in this chapter.) You can also use the **pr** command to format and print the contents of a file on your terminal:

```
Stardent 1500/3000: pr johnson↵


Mar  5 15:43 1986 johnson Page 1


March 5, 1986

Mr. Ron Johnson
Layton Printing
New York, N.Y.


Dear Mr. Johnson:

I enjoyed speaking with you this morning
about your company's plans to automate
your business.
Enclosed please find
the material you requested
about AB&C's line of computers.

If I can be of further assistance to you,
please don't hesitate to call.

Yours truly,

John Howe
.
.
Stardent 1500/3000:
```

The dots after the last line in the file represent the remaining lines (all blank in this case) that **pr** formatted into the output.

When the **pr** command is issued, the entire sixty-six lines print rapidly on your screen with no pause. In such cases, type CTRL S to interrupt the flow of printing on your screen. When you are ready to continue, type CTRL Q to resume printing. (Alternatively, you may pipe the output of **pr** to the **pg** command. See *Redirecting Output to a command.*)

See the **pr**(1) page in the *Command Reference Manual* for more on the **pr** command.

Requesting a Paper Copy of a File: the lp *Command*

The **lp** (short for line printer) command allows you to request a paper copy of a file from a printer.

To execute **lp**, follow this format:

lp *option(s) filename* ↵

In the following example a copy of the file *johnson* is requested.

```
Stardent 1500/3000: lp johnson↵
request id is laser-6885  (1 file)
Stardent 1500/3000:
```

The system responds with the name (or type) of the printer on which the file will be printed, and an identification (ID) number for your request. Here, the job is to be printed on a laser printer, has a request ID number of 6885, and includes one file.

To cancel a request to a printer type the **cancel** command and specify the request ID number. For example, to cancel your request for a printing of the file **letters** (request ID laser-6885) type:

cancel laser-6885 ↵

To check the status of a line printer job in progress or to get its request ID number issue the **lpstat** command. This command also gives a complete listing of every printer available on your system. See the **lp**(1) page and the **lpstat**(1) page in the *Commands Reference Manual* for a list of all available options.

Making a Duplicate Copy of a File: the **cp** *Command*

The **cp** (copy) command allows you to make a copy of a file while leaving the original intact. It is useful, for instance, if you want to modify a program you are writing while leaving the original program unchanged. The **cp** command also allows you to copy one or more files from one directory into another while leaving the original file or files in place.

The format of the **cp** command is as follows:

cp *file1 file2* ↵

The command requests that a copy of *file1* be made and placed in *file2*. The system returns a prompt when the copy is made.

In the following example the file *outline* is copied to a new file, *new.outline*. The **ls** command is then used to verify the existence of the new file.

```
Stardent 1500/3000: cp outline new.outline↵
Stardent 1500/3000: ls↵
new.outline
outline
table
Stardent 1500/3000:
```

Only one file in a directory can have a given name. In this case, because there was no file called *new.outline* when the **cp** command was issued, the system created a new file with that name. If a file called *new.outline* had already existed, its contents would have been replaced by the contents of outline.

CAUTION
Because of the danger of file overwriting, you should use the **ls** command to confirm the existence or non-existence of a file before issuing the **cp** command. (In the C-shell you can set the variable **noclobber**, which prevents accidental overwriting.)

If you had tried to copy *outline* to another file name *outline* in the same directory, it would not have worked. You would have gotten a message such as the one below.

```
Stardent 1500/3000: cp outline outline↵
cp: outline and outline are identical
Stardent 1500/3000: ls↵
outline
table
Stardent 1500/3000:
```

You can have two files with the same name as long as they are in different directories. For example, to copy the file *outline* from the *draft* directory to another file named *outline* in the *letters* directory, type the full pathname

cp outline /user1/starship/letters/outline ↵

or the relative pathname

cp outline ../letters/outline ↵

or

cp outline ../letters ↵

Note that (as in the last version of the command) it is sufficient to write the relative path to the directory *letters* without specifying the file name. The system automatically names the destination file *outline* unless another name is given.

Moving or Renaming a File: the **mv** *Command*

The **mv** (move) command allows you to rename a file in the same directory or to move a file from one directory to another. If you move a file to a different directory the file can be renamed or it can retain its original name.

To rename a file within one directory, follow this format:

mv *file1 file2* ↵

The **mv** command changes a file's name from *file1* to *file2* and deletes *file1*. Remember that the names *file1* and *file2* can be any valid names, including pathnames.

CAUTION
As with the **cp** command, the **mv** command will overwrite existing files (unless the C-shell variable **noclobber** has been set). Check filenames with the **ls** command before moving or renaming a file.

For example, if you are in the directory *draft* in the sample file system and you would like to change the name of file *table* to *new.table*, simply type

mv table new.table ↵

If the command executes successfully you receive a prompt. To verify that the file *new.table* exists use the **ls** command to list the contents of the directory. The screen shows your input and the system's output as follows:

```
Stardent 1500/3000: mv table new.table↵
Stardent 1500/3000: ls↵
new.table
outline
Stardent 1500/3000:
```

You can also move a file from one directory to another keeping the same name or changing it to a different one. To move the file without changing its name, use the following command line:

mv *file(s) directory* ↵

The file and directory names can be any valid names, including pathnames.

For example, if you want to move the file *table* from the current directory named *draft* (whose full pathname is */user1/starship/draft*) to a file with the same name in the directory *letters* (whose relative pathname from *draft* is *../letters* and whose full pathname is */user1/starship/letters*), use any of several command lines:

mv table /user1/starship/letters ↵

mv table /user1/starship/letters/table ↵

mv table ../letters ↵

mv table ../letters/table ↵

Now suppose you want to change the name of file *table* to *table2* while moving it to the directory *letters*. Use either of these command lines:

mv table /user1/starship/letters/table2 ↵

mv table ../letters/table2 ↵

You can verify that the command worked by using the **ls** command to list the contents of the directory.

Removing a File: the rm *Command*

When you no longer need a file, you can remove it from your directory by executing the **rm** (short for remove) command:

rm *file(s)* ↵

You can remove more than one file at a time by including the filenames as arguments on the command line:

rm *file1 file2 file3* ↵

The system does not save a copy of a file it removes; once you have executed this command, your file is removed permanently.

After you have issued the **rm** command, you can verify its successful execution by running the **ls** command. Since **ls** lists the files in your directory, you'll immediately be able to see whether or not **rm** has executed successfully.

For example, say you have a directory that contains two files, *outline* and *table*. You can remove both files by issuing the **rm** command once. If **rm** is executed successfully, your directory will be empty. Verify this by running the **ls** command.

```
Stardent 1500/3000: rm outline table↵
Stardent 1500/3000: ls↵
Stardent 1500/3000:
```

The prompt shows that *outline* and *table* were removed.

To remove multiple files in a directory use the special pattern matching characters *, ?, or [ ] on the command line. This is a powerful capability; please refer to *Pattern Matching* before attempting to use it.

Counting Lines, Words, and Characters in a File: the wc *Command*

The **wc** (word count) command reports the number of lines, words, and characters there are in the file(s) named on the command line. If you name more than one file the **wc** program counts the number of lines, words, and characters in each specified file and then totals the counts. If you wish, you can direct the **wc** command to give you only a line, a word, or a character count by using the **–l**, **–w**, or **–c** options, respectively.

The general format for the **wc** command is as follows:

wc *option(s) file(s)* ↵

Specifically, if you issue the command

wc file1 ↵

The system responds with a line in the following format:

l w c file1

l Represents the number of lines in *file1*.

w Represents the number of words in *file1*.

c Represents the number of characters in *file1*.

For example, to count the lines, words, and characters in the file *johnson* (located in the current directory, *letters*) type the following command line:

```
Stardent 1500/3000: wc johnson↵
24      66      406 johnson
Stardent 1500/3000:
```

The system response means that the file *johnson* has 24 lines, 66 words, and 406 characters.

You can use the **wc** command to count the lines, words, and characters in more than one file. The following example shows the counts of lines, words, and characters in the files *johnson* and *sanders* in the current directory.

```
Stardent 1500/3000: wc johnson sanders↵
     24      66      406 johnson
     28      92      559 sanders
     52     158      965 total
Stardent 1500/3000:
```

The first column gives the number of lines (28 for sanders), the second gives words (92 for sanders), and the third column gives characters (559 for sanders). As you can see, the last line totals the columns in the first two lines.

To get only a line, a word, or a character count, use the **–l**, **–w**, or **–c** options, respectively.

For example, if you use the –l option, the system reports only the number of lines in *sanders*.

```
Stardent 1500/3000: wc –l sanders↵
      28 sanders
Stardent 1500/3000:
```

Protecting Your Files: the chmod *Command*

The **chmod** (change mode) command allows you to decide who can see and use your files and directories and who cannot. The following three symbols are used for assigning permissions.

r Allows system users to read a file or to copy its contents.

w Allows system users to write changes into a file (or a copy of it).

x Allows system users to run an executable file or search a directory.

To specify the users to whom you are granting (or denying) these types of permission, use the following symbols.

u You, the owner of the files and directories (**u** is short for user).

g Members of the group to which you belong. (The group could consist of team members working on a project, members of a department, or a group arbitrarily designated by the person who registered you as a user. Please see the *System Administrator's Guide* for details about defining groups.)

o All other system users.

Your system has been set up so that when you create a file or a directory the system automatically grants or denies permission to you, members of your group, and other system users. As the owner of the file or directory you always have the option of changing these permissions. For example, you may want to keep certain files private and reserve them for your exclusive use. You may want to grant permission to read and write changes into a file to members of your group and all other system users as well. Or you may share a program with members of your group by granting them permission to execute it.

How to Determine Existing Permissions

You can determine what permissions are currently in effect on a file or a directory by using the **ls** command with the –l option.

```
Stardent 1500/3000: ls -l↵
total 35
-rwxr-xr--   1 starship      project    9346  Nov 1  08:06  display
-rw-r--r--   1 starship      project    6428  Dec 2  10:24  list
drwx--x--x   2 starship      project      32  Nov 8  15:32  tools
Stardent 1500/3000:
```

Permissions for the *display* and *list* files and the *tools* directory are shown on the left of the screen under the line

```
total 35
```

and appear in this format:

`-rwxr-xr--` (for the **display** file)
`-rw-r--r--` (for the **list** file))
`drwx--x--x` (for the **tools** directory)

After the initial character that describes the file type (for example a - (dash) symbolizes a regular file and a `d` a directory), the other nine permission-setting characters consist of three sets of three characters. The first set refers to permissions for the owner, the second set to permissions for group members, and the last set to permissions for all other system users. Within each set of characters, the `r`, `w`, and `x` show the permissions currently granted to each category. If a dash appears instead of an `r`, `w`, or `x`, permission to read, write, or execute is denied.

Here is the breakdown for the file named *display*.

Permissions for the File *display*

-	r w x	r - x	r - -
regular file	user can read, write, execute	group can read, execute	others can read

As you can see, the owner has `r`, `w`, and `x` permissions, members of the group have `r` and `x` permissions, and others have `r` permission only.

There are two exceptions to this notation system. Occasionally the letter `s` or the letter `l` may appear in the permissions line instead of an `r`, `w` or `x`. The letter `s` (short for set user ID or set group ID) represents a special type of permission to execute a file. It appears where you normally see an `x` (or –) for the user or group (the first and second sets of permissions). From a user's point of view it is equivalent to an `x` in the same position; it implies that execute permission exists. It is significant only for programmers and system administrators. (See the *System Administrator's Guide* for details about setting the user or group ID.)

The letter l is the symbol for lock enabling (for special security reasons). It does not mean that the file has been locked. It simply means that the function of locking is enabled, or possible, for this file. The file may or may not be locked; that cannot be determined by the presence or absence of the letter `l`.

How to Change Existing Permissions. After you have determined what permissions are in effect you can change them by executing the **chmod** command in the following format:

chmod *who+permission file(s)* ↵

or

chmod *who–permission file(s)* ↵

The following list defines each component of this command line.

chmod	Is the name of the program.
who	Is one of three user groups (**u**, **g**, or **o**). **u** = user **g** = group **o** = others
+ or –	Is the instruction that grants (+) or denies (–) permission.
permission	Is any combination of three authorizations (**r**, **w**, and **x**). **r** = read **w** = write **x** = execute
file	Is the file or directory name.

The **chmod** command does not work if you type a space between *who*, the instruction that gives (+) or denies (–) permission, and the *permission*.

The following examples show a few ways to use the **chmod** command. As the owner of *display*, you can read, write, and run this executable file. You can protect the file against being accidentally changed by denying yourself write (**w**) permission. To do this, type the command line:

chmod u–w display ↵

After receiving the prompt use the **ls –l** command to verify that this permission has been changed, as shown in the following screen.

```
Stardent 1500/3000: chmod u-w display↵
Stardent 1500/3000: ls -l↵
total 35
-r-xr-xr--   1 starship      project    9346  Nov 1  08:06  display
rw-r--r--    1 starship      project    6428  Dec 2  10:24  list
drwx--x--x   2 starship      project      32  Nov 8  15:32  tools
Stardent 1500/3000:
```

As you can see, you no longer have permission to write changes into the file. You will not be able to change this file until you restore write permission for yourself.

If you omit the *who* in the chmod command, the change is implemented for all three user groups (**u, g, o**). For instance:

chmod +x list ↵

adds execute permission to the file *list* for all three groups.

Now consider another example. Notice that permission to write into the file *display* has been denied to members of your group and other system users. They do have read permission, however. This means they can copy the file into their own directories and then make changes to it. To prevent all system users from copying this file, you can deny them read permission by typing

chmod go–r display ↵

The **g** and **o** stand for group members and all other system users, respectively, and the **–r** denies them permission to read or copy the file. Check the results with the **ls –l** command.

```
Stardent 1500/3000: $ chmod go-r display↵
Stardent 1500/3000: ls -l↵
total 35
-rwx--x--x   1 starship      project    9346  Nov 1  08:06  display
-rw-r--r--   1 starship      project    6428  Dec 2  10:24  list
drwx--x--x   2 starship      project      32  Nov 8  15:32  tools
Stardent 1500/3000:
```

An Alternative Method of Changing Permissions. The **chmod** command accommodates two methods for changing permissions. The method described above in which symbols such as `r`, `w`, and `x` are used to specify permissions is called the symbolic method.

The alternative method is called the octal (or base eight) method. Its format requires you to specify permissions using numbers ranging from 0 to 7 (octal numbers).

With the octal method the **chmod** command has the following format:

chmod *UGO filename* ↵

where you the user substitutes a numeric code for each of the letters U, G and O. (U represents the code for user, G for group, and O for other.) You compute the code by adding the following values:

```
0       for no permission
4       for read permission
2       for write permission
1       for execute permission
```

For example, for the file *display*, to grant read, write and execute permission to yourself, read and write permission to your group, and read permission to others, you issue the command

chmod 764 *display* ↵

For the file *list*, to give yourself read and execute permission, your group execute permission and others no permission, you issue the command:

chmod 510 *list* ↵

Here you can see the results of these commands.

```
Stardent 1500/3000: ls -l display list↵
total 35
-rwxrw-r--   1 starship      project     9346  Nov 1  08:06  display
-r-x--x---   1 starship      project     6428  Dec 2  10:24  list
Stardent 1500/3000:
```

To learn more about the octal method see the **chmod**(1) page in the *Commands Reference Manual.*

A Note on Permissions and Directories

You can use the **chmod** command to grant or deny permission for directories as well as files. Simply specify a directory name instead of a file name on the command line.

You should, however, consider the impact on other users of changing permissions for directories. For example, say you grant read permission for a directory to yourself (**u**), members of your group (**g**), and other system users (**o**). Every user who has access to the system is able to read the names of the files contained in that directory by running the **ls –l** command. Similarly, granting write permission allows the designated users to create new files in the directory and remove existing ones. Granting permission to execute the directory allows designated users to use the **cd** command to move to that directory (and make it their current directory).

Identifying Differences Between Files: the **diff** *Command*

The **diff** command locates and reports all differences between two files and tells you how to change the first file so that it is a duplicate of the second. The basic format for the command is

diff *file1 file2* ↵

If *file1* and *file2* are identical, the system returns a prompt to you. If they are not, the **diff** command instructs you on how to change the first file so it matches the second. The command flags lines in *file1* (to be changed) with the < (less than) symbol and lines in *file2* (the model text) with the > (greater than) symbol.

For example, say you execute the **diff** command to identify the differences between the files *johnson* and *mcdonough*. The *mcdonough* file contains the same letter that is in the *johnson* file, with appropriate changes for a different recipient. The **diff** command identifies those changes as follows:

```
3,6c3,6
< Mr. Ron Johnson
< Layton Printing
< 52 Hudson Street
< New York, N.Y.
---
> Mr. J.J. McDonough
> Ubu Press
> 37 Chico Place
```

```
> Springfield, N.J.
9c9
< Dear Mr. Johnson:
---
> Dear Mr. McDonough:
```

The first line of output from **diff** is

```
3,6c3,6
```

This means that if you want *johnson* to match *mcdonough*, you must change (c) lines 3 through 6 in *johnson* to lines 3 through 6 in *mcdonough*. The **diff** command then displays both sets of lines.

If you make these changes (using a text editor such as **vi**), the *johnson* file becomes identical to the *mcdonough* file. Remember, the **diff** command identifies differences between specified files. If you want to make an identical copy of a file, use the **cp** command. See the **diff**(1) page in the *Commands Reference Manual* for more on the command and its options.

Searching a File for a Pattern: the **grep** Command

You can instruct the operating system to search through a file for a specific word, phrase, or group of characters by executing the **grep** command. (The word **grep** is short for globally search for a regular expression and print. A regular expression is any pattern of characters, such as a word, a phrase, or an equation.)

The basic format for the command line is:

grep *pattern file(s)* ↵

The following example shows the lines that contain the word automation in the file *johnson*.

```
Stardent 1500/3000: grep automation johnson ↵
and office automation software.
Stardent 1500/3000:
```

The output consists of all the lines in the file *johnson* that contain the pattern for which you were searching (automation).

If the pattern contains multiple words or any character that conveys special meaning to the operating system, (such as **$**, **|** , *****, **?**, and so on), you must enclose the entire pattern in single or double

quotes. For example, if you want to locate the lines containing the pattern *office automation*, type

```
Stardent 1500/3000: grep 'office automation' johnson ↵
and office automation software.
Stardent 1500/3000:
```

If you cannot recall which letter contained a reference to office automation, your letter to Mr. Johnson or the one to Mrs. Sanders, type the following command line.

```
Stardent 1500/3000: grep 'office automation' johnson sanders ↵
johnson:and office automation software.
Stardent 1500/3000:
```

The output tells you that the pattern *office automation* is found once in the *johnson* file.

In addition to the **grep** command, the operating system provides variations of it called **egrep** and **fgrep**, along with several options that enhance the searching powers of the command. See the **grep**(1), **egrep**(1), and **fgrep**(1) pages in the *Commands Reference Manual* for complete explanation of these commands.

Sorting and Merging Files: the **sort** *Command*

The operating system provides an efficient tool called **sort** for sorting and merging files. The format for the command line is:

sort0 *option(s) file(s)* ↵

This command causes lines in the specified files to be sorted and merged in the following order.

- Lines beginning with numbers are sorted by digit and listed before lines beginning with letters.
- Lines beginning with upper case letters are listed before lines beginning with lower case letters.
- Lines beginning with symbols such as *, %, or @, are sorted on the basis of the symbol's ASCII representation.

For example, let's say you have two files, *group1* and *group2*, each containing a list of names. You want to sort each list

alphabetically and then interleave the two lists into one. First, display the contents of the files by executing the **cat** command on each.

```
Stardent 1500/3000: cat group1↵
Smith, Allyn
Jones, Barbara
Cook, Karen
Moore, Peter
Wolf, Robert
Stardent 1500/3000: cat group2↵
Frank, M. Jay
Nelson, James
West, Donna
Hill, Charles
Morgan, Kristine
Stardent 1500/3000:
```

Now sort and merge the contents of the two files by executing the **sort** command.

```
Stardent 1500/3000: sort group1 group2↵
Cook, Karen
Frank, M. Jay
Hill, Charles
Jones, Barbara
Moore, Peter
Morgan, Kristine
Nelson, James
Smith, Allyn
West, Donna
Wolf, Robert
Stardent 1500/3000:
```

In addition to combining simple lists as in the example, the **sort** command can be used to rearrange lines and parts of lines (called fields) according to a number of other specifications you designate on the command line. Refer to the *Commands Reference Manual* for a full description of the **sort** command and its available options.

B-2. Basics: The Shell

The shell is a powerful command interpreter that provides the interface between you and the Stardent 1500/3000 operating system, and allows you to do tasks such as managing files and grouping commands together for streamlined execution. This section shows you how to use the shell command language, write

simple shell programs, and modify your shell login environment.

An enhanced version of the shell, the C-shell, can be used as an alternative command interface and programming language. This section contains a brief description of the C-shell (see *The C-Shell*, below).

Shell Command Language

The following paragraphs introduce commands and characters with special meanings that let you

- Find and manipulate a group of files using pattern matching.
- Run a command in the background or at a specified time.
- Run a group of commands sequentially.
- Redirect standard input and output from and to files and other commands.
- Terminate processes.

Pattern Matching

Pattern matching characters, or wild cards, are used to represent file names or parts of file names, thereby simplifying commands that use file names as arguments. The following pattern matching characters are discussed in the next paragraphs.

* (asterisk)	Matches all characters.
? (question mark)	Matches any single character.
[ ] (brackets)	Matches any of a specified set of characters.

Matching All Characters: the Asterisk (*). The asterisk (*) matches any string of characters, including a null (empty) string, and can be used to to specify a full or partial file name. * alone refers to all the file and directory names in the current directory. For example, the command

ls * ↵

lists all of the files in your current directory.

CAUTION
The * is a powerful character. If you type **rm *** you erase all the files in your current directory. Be very careful how you use it!

Suppose you have written several reports, with names *report*, *report1*, *report1a*, *report1b.01*, *report25*, and *report216*. To find out how many reports you have written, use the **ls** command to list all files that begin with the string *report*.

```
Stardent 1500/3000: ls report* ↵
report
report1
report1a
report1b.01
report25
report316
Stardent 1500/3000:
```

The * matches any characters after the string *report*, including no letters at all.

The * can represent characters in any part of the file name, and can be used any number of times in a command line. For example, if you know that several files have a the letter *F* in common, you can request a list of them on that basis.

ls *F* ↵

The system responds as follows.

```
123F
F
FATE
Fig3.4E
```

The order is determined by the ASCII sort sequence: (1) numbers; (2) upper case letters; (3) lower case letters.

Matching One Character: the Question Mark (?). The question mark (?) matches any single character of a file name. Let's say you have written several chapters in a book that has twelve chapters, and you want a list of those you have finished through Chapter 9. Use the **ls** command with the **?** to list all chapters that begin with the string *chapter* and end with any single character, as shown below:

```
Stardent 1500/3000: ls chapter? ↵
chapter1
chapter2
chapter5
chapter9
Stardent 1500/3000:
```

The system responds by printing a list of all file names that match.

Matching One of a Set of Characters: Brackets ([ ]). Use brackets ([ ]) to match any one of several possible characters that may appear in one position in the file name. For example, if you include **[crf]** as part of a file name pattern, the shell looks for file names that have the letter **c**, the letter **r**, or the letter **f** in the specified position.

```
Stardent 1500/3000 :ls [crf]at ↵
cat
fat
rat
Stardent 1500/3000:
```

This command displays all three-letter file names that begin with the letter **c**, **r**, or **f** and end with the letters **at**. Characters that are grouped within brackets in this way are collectively called a character class.

Brackets can also be used to specify a range of characters. For example, if you specify

chapter[1-5]

the shell matches any files named *chapter1* through *chapter5*. This is an easy way to handle only a few chapters at a time. If you specify

pr chapter[2-4] ↵

The shell prints the contents of *chapter2*, *chapter3*, and *chapter4*, in that order, on your terminal.

You may also use a character class to specify a range of letters. If you specify **[A-Z]**, the shell looks only for upper case letters; if **[a-z]**, only lower case letters.

Special Characters

The shell language has other special characters that perform useful functions. Some of these are discussed in the following paragraphs; others are described in *Input and Output Redirection*, later in this chapter.

Running a Command in Background: the Ampersand (&). Some shell commands take considerable time to execute. The ampersand (&) is used to execute commands in background mode, freeing your terminal for other tasks. The general format for running a command in background mode is

command **&** ↵

In the next example, a search for the string *delinquent* in the file *accounts* is run in the background.

NOTE
You should not run interactive shell commands, such as **read**, in the background.

```
Stardent 1500/3000: grep delinquent accounts &↵
21940
Stardent 1500/3000:
```

When you run a command in the background the operating system outputs a process number; `21940` is the process number in the example. You can use this number to stop the execution of a background command. (Stopping the execution of processes is discussed under *Executing and Terminating Processes* later in this chapter.) The prompt on the last line means the terminal is free and waiting for your commands; **grep** has started running in background.

Running a command in the background affects only the availability of your terminal; it does not affect the output of the command. Whether or not a command is run in background, it prints its output on your terminal screen unless you redirect it to a file. (See *Input and Output Reduction*, later in this chapter, for details.)

If you want a command to continue running in the background after you log off, you can submit it with the **nohup**(1) command. (This is discussed in *Using the* **nohup** Command later in this chapter.)

Executing Commands Sequentially: the Semicolon (;). You can type two or more commands on one line as long as each pair is separated by a semicolon (;), as follows:

command1; *command2*; *command3* ↵

The operating system executes the commands in the order that they appear in the line and prints all output on the screen. This process is called sequential execution. For example, if you enter

cd; pwd; ls ↵

The shell executes these commands sequentially:

(1) **cd** changes your location to your home directory.

(2) **pwd** prints the full pathname of your home directory.

(3) **ls** lists the files in your home directory.

If you do not want the system's responses to these commands to appear on your screen, refer to *Input and Output Redirection*, later in this chapter, for instructions.

Turning Off Special Meanings: the Backslash (\). The shell interprets the backslash (\) as an escape character that allows you to turn off any special meaning of the character immediately after it. For example, if you want to use the **grep** command to search a file named *trial* for an *, type

```
Stardent 1500/3000: grep \* trial↵
The all * game
Stardent 1500/3000:
```

The **grep** command finds the * in the text and displays the line in which it appears. Without the \ the * would appear as a metacharacter to the shell and would match all lines in the file.

For a complete list of special characters understood by the shell see *Using Special Characters as Literal Characters*, Chapter 1.

Turning Off Special Meanings: Quotes. Another way to escape the meaning of a special character is to use quotation marks. Single quotes ('...') turn off the special meaning of any character. Double quotes ("...") turn off the special meaning of all characters except $ and ` (grave accent), which retain their special meanings within double quotes. An advantage of using quotes is that numerous special characters can be enclosed in the quotes; this can be more concise than using the backslash.

For example, if your file named *trial* contains the line

```
He really wondered why. Why???
```

use the **grep** command to find the line with the three question marks as follows:

```
Stardent 1500/3000: grep '???' trial ↵
He really wondered why. Why???
Stardent 1500/3000:
```

If you had omitted the quotes, the three question marks would have been interpreted as shell metacharacters and matched all file names of three characters.

Using Quotes to Turn Off the Meaning of a Space. A common use of quotes as escape characters is to turn off the special meaning of the blank space. The shell interprets a space on a command line as a delimiter between the arguments of a command. Both

single and double quotes allow you to escape that meaning.

For example, to locate two or more words that appear together in text, make the words a single argument (to the **grep** command) by enclosing them in quotes.

```
Stardent 1500/3000: grep 'The all' trial ↵
The all * game
Stardent 1500/3000:
```

Input and Output Redirection

Many shell commands expect to receive their input from the keyboard (standard input) and most commands display their output on the screen (standard output). Redirection is used to reassign the standard input and output to other files or programs. With redirection, you can tell the shell to

- Take its input from a file rather than the keyboard.
- Send its output to file rather than the screen.
- Use a program as the source of data for another program.
- Supply lines of input to a command, where the command is contained within a program.

You use a set of operators, the less than sign (<), the greater than sign (>), two greater than signs (>>), two less than signs (<<), and the pipe symbol (|) to redirect input and output.

Redirecting Input: the < Sign

To redirect input, specify a file name after a less than sign (<) on the command line:

command < *file* ↵

For example, assume that you want to use the **mail** command (described in chapter 6) to send a file named *report* to another user with the login *colleague*. Specify the file name as the source of input:

mail colleague < report ↵

Redirecting Output to a File: the > Sign

To redirect output, specify a file name after a greater than sign (>) on the command line:

command > *file* ↵

Before redirecting the output of a command to a particular file, use the **ls** command to make sure that a file by that name does not already exist (unless you don't care if you lose the file). Because the shell does not allow you to have two files of the same name in a directory, the shell overwrites the contents of the existing file with the output of your command if you redirect the output to a file with the existing file's name. The shell does not warn you about overwriting the original file.

CAUTION
If you redirect output to a file that already exists, the output of your command overwrites the contents of the existing file.

Appending Output to an Existing File: the >> Symbol

To keep from destroying an existing file, you can also use the double greater than sign symbols (>>), as follows:

command >> *file* ↵

This appends the output of a command to the end of *file*. If *file* does not exist, it is created.

The following example shows how to append the output of the **cat** command to an existing file. First, the **cat** command is executed on both files without output redirection to show their respective contents. Then the contents of *file2* are added after the last line of *file1* by executing the **cat** command on *file2* and redirecting the output to *file1*.

```
Stardent 1500/3000: cat file1↵
Now is the time
for all
good men
Stardent 1500/3000:
Stardent 1500/3000: cat file2↵
to come to the aid
of their country.
Stardent 1500/3000:
Stardent 1500/3000: cat file2 >> file1↵
Stardent 1500/3000: cat file1↵
Now is the time
for all
good men
to come to the aid
of their country.
Stardent 1500/3000:
```

Useful Applications of Output Redirection

Redirecting output is useful when you do not immediately want output to appear on your screen or when you want to save it. Output redirection is also especially useful when you run commands that perform clerical chores on text files. Two such commands are **spell** and **sort**.

In this example **spell** searches a file named *memo* and places a list of misspelled words in a file named *misspell*. If the output of **spell** were not redirected to a file, a list of misspelled words would print on the screen.

spell memo > misspell ↵

Combining Input and Output Redirection

Input and Output redirection can be used together in a command, allowing you to take input from one file and send output to another file. The general format is

command < *input_file* > *output_file* ↵

Combining Background Mode and Output Redirection

Running a command in the background does not affect the command's output; unless the output is redirected, output is always printed on the screen. If you are using your terminal to perform other tasks while a command runs in the background, you are interrupted when the command displays its output on your screen. If you redirect that output to a file, however, you can work undisturbed.

For example, in the *Special Characters* section you learned how to execute the **grep** command in the background with &. Now suppose you want to find occurrences of the word *test* in a file named *schedule*. Run the **grep** command in the background and redirect its output to a file called *testfile*:

```
Stardent 1500/3000: grep test schedule > testfile &↵
```

You can then use your terminal for other work and examine *testfile* when you have finished.

Supplying Lines of Input to a Command: the << Symbol

The shell allows you to supply lines of input to a command, where the command is contained within a shell program. It is a way to supply input to a program without using a separate input file. (The lines of input are sometimes called a "here document"). The notation consists of the redirection symbol << and a delimiter that specifies the beginning and end of the lines of input. The exclamation mark (!) is generally used as the delimiter. To redirect input lines you use a special command line including the << and ! signs, followed by the input lines, followed by a terminating !.

command <<! ↵ *...input lines...* ↵ ! ↵

In the next example, the program **gbday** uses input line redirection to send a generic birthday greeting by redirecting lines of input into the **mail** command:

```
Stardent 1500/3000: cat gbday↵
mail -s 'Birthday Greeting' $1 <<!
Best wishes to you on your birthday.
!
Stardent 1500/3000:
```

The input line to the mail command is

```
Best wishes to you on your birthday
```

When you use this command, you must specify the recipient's login as the argument to the command. For example, to send this greeting to the owner of login **mary**, type

gbday mary ↵

Login **mary** will receive your greeting the next time she reads her mail messages:

```
Message 1
From mylogin Wed May 14 14:31 CDT 1986
To: mary
Subject:  Birthday Greeting
Status: R

Best wishes to you on your birthday
```

For more on the **mail** command see Chapter 5.

Redirecting Output to a Command: the Pipe (|)

The | character is called a pipe. Pipes are powerful tools that take the output of one command and use it as input for another command without creating temporary files. A multiple command line created in this way is called a pipeline.

The general format for a pipeline is

command1 | *command2* | *command3...* ↵

The output of *command1* is used as the input of *command2*. The output of *command2* is then used as the input for *command3*.

To understand the efficiency and power of a pipeline, consider the contrast between two methods that achieve the same results.

- To use the input/output redirection method, run one command and redirect its output to a temporary file. Then run a second command that takes the contents of the temporary file as its input. Finally, remove the temporary file after the second command has finished running.

- To use the pipeline method, run one command and pipe its output directly into a second command.

For example, say you want to mail a happy birthday message in a banner to the owner of the login **david**. Doing this without a pipeline is a three-step procedure. You must

(1) Enter the **banner** command and redirect its output to a temporary file:

banner happy birthday > message.tmp

(2) Enter the **mail** command using **message.tmp** as its input:

mail david < message.tmp

(3) Remove the temporary file:

rm message.tmp

However, by using a pipeline you can do this in one step:

banner happy birthday | mail david

Using the pipe along with the **pg** command is an excellent way to view output from a program without having to redirect the output to a file. For example, to view the misspelled words in the file *memo*, type:

spell memos | pg ↵

You then have the capabilities of the **pg** command for perusing the list of misspelled words. For a review of the **pg** command see *Paging Through the Contents of a File.*

Substituting Output for an Argument

The output of any command may be captured and used as arguments on a command line. This is done by enclosing the command in grave accents (`...`) and placing it on the command line in the position where the output should be treated as arguments. This is known as command substitution.

For example, you can substitute the output of the **date** command for the argument in a **banner** printout by typing the following command line:

```
Stardent 1500/3000: banner`date` ↵
```

The system prints a banner with the current date and time.

The *Shell Programming* section in this chapter shows you how you can also use the output of a command line as the value of a variable.

Executing and Terminating Processes

This section discusses the following commands for controlling processes:

batch or **at**	To schedule commands for a later time.
ps	To obtain the status of active processes.
kill	To terminate active processes.
nohup	To keep background processes running after you have logged off.

Running Commands at a Later Time With the batch *and* at *Commands*

The **batch** and **at** commands 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 CTRL d.

NOTE
The character combination CTRL d means "End of File" and is generally used to terminate input or to log off the system. An exception is its special meaning to the vi editor.

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.

The general format for **batch** is

batch ↵
first command ↵
.
.
.
last command ↵
CTRL d

If there is only one command to be run with **batch**, you can enter it as follows:

batch *command_line* ↵
CTRL d

The next example uses **batch** to execute the **grep** command at a convenient time. Here **grep** searches all files in the current directory and redirects the output to the file *dol.file.*

```
Stardent 1500/3000: batch grep dollar * > dol-file↵
CTRL d
job 155223141.b at Sun Dec 7 11:14:54 1986
Stardent 1500/3000:
```

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.

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 ↵
CTRL d

The *time* argument consists of the time of day and, if the date is not today, the date. You have choices of syntax for the date; see **at**(1) in the *Commands Reference Manual* for specifics.

The following example shows how to use the **at** command to mail a happy birthday **banner** to login **emily** on her birthday:

```
Stardent 1500/3000: at 8:15am Feb 27↵
banner happy birthday | mail -s 'Happy Birthday' emily ↵
CTRL d
job 453400603.a at Thurs Feb 27 08:15:00 1986
Stardent 1500/3000:
```

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* ↵

If you have forgotten the job number, the **at –l** command gives you a list of the current jobs in the **batch** or **at** queue, as the following screen shows:

```
Stardent 1500/3000: at -l↵
user = mylogin 168302040.a at Sat Nov 29 13:00:00 1986
user = mylogin 453400603.a at Fri Feb 27 08:15:00 1987
Stardent 1500/3000:
```

Notice that the system displays the job number and the time the job will run.

Obtaining the Status of Running Processes

The **ps** (process status) command gives you the status of all the processes you are currently running. For instance, use the **ps** command to show the status of all processes that you are running in the background using **&** (described earlier in *Special Characters*).

In the following example, **grep** is run in the background, and then the **ps** command is issued. The system responds with the process identification (PID) and the terminal identification (TTY) number. It also gives the cumulative execution time for each process (TIME), and the name of the command that is being executed (COMMAND).

```
Stardent 1500/3000:  grep word * > temp & ↵
28223
Stardent 1500/3000:
Stardent 1500/3000: ps↵
PID     TTY      TIME COMMAND
28124   tty10    0:00   sh
28223   tty10    0:04   grep
28224   tty10    0:04   ps
Stardent 1500/3000:
```

Notice that the system reports a PID number for the **grep** command, as well as for the other processes that are running: the **ps** command itself, and the **sh** (shell) command that runs while you are logged in.

Terminating Active Processes

The **kill** command is used to terminate active shell processes that are running in the background. (The command CTRL C is used to terminate commands running in the foreground.) The general format for the **kill** command is

kill *PID* ↵

where *PID* is the process identification.

The following example shows how you can terminate the **grep** command that you started executing in background in the previous example.

```
Stardent 1500/3000:  kill 28223↵
28223 Terminated
Stardent 1500/3000:
```

Notice the system responds with a message and a Stardent 1500/3000 prompt, showing that the process has been killed. If the system cannot find the PID number you specify, it responds with an error message:

```
kill:28223:No such process
```

Using the **nohup** *Command*

Processes are automatically killed when you log off. If you want a background process to continue running after you log off, you must use the **nohup** command to run the process.

nohup *command* **&** ↵

Notice that you place the **nohup** command before the command you intend to run as a background process.

For example, say you want the **grep** command to search all the files in the current directory for the string *word* and redirect the output to a file called *word.list*, and you wish to log off immediately afterward. Type the command line as follows:

nohup grep word * > word.list & ↵

You can terminate the **nohup** command by using the **kill** command.

Shell Programming

This section shows you how to create and execute shell programs containing commands, variables, positional parameters, return codes, and basic programming control structures. Shell programs allow you to streamline and simplify your interactions with Stardent 1500/3000.

Creating and Executing a Simple Shell Program

To create a simple shell program that

- Prints the current directory.
- Lists the contents of that directory.
- Displays this message on your terminal: "This is the end of the shell program."

simply create a file called *dir_list* and enter these three lines into it:

pwd ↵
ls ↵
echo This is the end of the shell program. ↵

One way to execute this program is to use the **sh** command. If you type

sh dir_list ↵

sh executes the **dir_list** command. It first prints the pathname of the current directory, then the list of files in the current directory,

and finally, the comment `This is the end of the shell program.`

If **dir_list** is a useful command, use the **chmod** command to make it an executable file; then you can type **dir_list** by itself to execute the commands it contains. The following example shows how to use the **chmod** command to make a file executable and then runs the **ls –l** command to verify the changes you have made in the permissions.

```
Stardent 1500/3000: chmod u+x dir_list↵
Stardent 1500/3000: ls -l↵
total 2
-rw-------      1   login   login    3661   Nov  2 10:28 mbox
-rwx------      1   login   login      48   Nov 15 10:50 dir_list
Stardent 1500/3000:
```

Notice that **chmod** turns on permission to execute (**+x**) for the user (**u**). Now **dir_list** is an executable program and can be executed by typing

dir_list ↵

Creating a bin *Directory for Executable Files*

You can make your shell programs executable from all your directories, by creating a special directory, often called *bin*, moving your shell programs to it, and setting your **PATH** environment variable to include the directory.

To create a *bin* directory and move **dir_list** to it, type

cd ↵
mkdir bin ↵
mv dir_list bin/dir_list ↵

To set your environment variable **PATH** to include your **bin** directory, add the following line to your .profile (located in your home directory):

PATH=$PATH:$HOME/bin

(If the **PATH** variable has already been defined in your **pathname** **$HOME/bin** to the definition. If you have any questions or problems please see *Modifying Your Login Environment* later in this chapter. See *Variables* below for more information about **PATH**.)

Once you have completed this process, **dir_list** and any other shell programs you move to your bin directory are executable from any of your directories.

Warnings about Naming Shell Programs

You can give your shell program any appropriate file name. However, you should not give your program the same name as a system command. If you do, the shell executes your command instead of the system command. For example, if you had named your **dir_list** program **mv**, each time you tried to move a file, the system would have executed your directory list program instead of **mv**.

Variables

Shell programs use three basic types of variables that are described in this section.

- Positional parameters.
- Special parameters.
- Named variables.

Positional Parameters

A positional parameter is a variable within a shell program whose value is set from an argument specified on the command line invoking the program. Positional parameters are numbered and are referred to with a preceding **$**: **$1, $2, $3,** and so on.

a shell program may refer to up to nine positional parameters. If a shell program is invoked with the following command line,

shell.prog param1 param2 param3 ↵

then positional parameter **$1** within the program is assigned the value *param1,* positional parameter **$2** within the program is assigned the value *param2,* and positional parameter **$3** is assigned the value *param3* when the shell program is invoked. The shell program itself is referred to as $0.

The following example

(1) Shows a shell program named **pp** containing three echo commands with positional parameters.

(2) Uses the **chmod** command to make **pp** executable.

(3) Executes the **pp** command with the arguments **one, two,** and **three.**

```
Stardent 1500/3000: cat pp↵
echo  The first positional parameter is: $1
echo  The second positional parameter is: $2
echo  The third positional parameter is: $3
echo  The fourth positional parameter is: $4
Stardent 1500/3000: chmod u+x pp↵
Stardent 1500/3000: pp one two three four↵
The first positional parameter is: one
The second positional parameter is: two
The third positional parameter is: three
The fourth positional parameter is: four
Stardent 1500/3000:
```

The following example shows the shell program **bbday,** which mails a greeting to the login entered in the command line.

```
Stardent 1500/3000: cat bbday↵
banner happy birthday | mail 'Happy Birthday' $1
```

Try sending yourself a birthday greeting. If your login name is **sue,** your command line will be:

bbday sue ↵

The shell allows a command line to contain 128 arguments. However, a shell program is restricted to at most nine positional parameters, **$1** through **$9,** at a given time.

Special Parameter: $#

The **$#** parameter within a shell program contains the number of arguments with which the shell program was invoked. Its value can be used anywhere within the shell program.

In the following example, the executable shell program called **get.num** contains one line.

```
Stardent 1500/3000: cat get.num↵
echo The number of arguments is: $#
Stardent 1500/3000:
```

get.num simply displays the number of arguments with which it is invoked. For example

```
Stardent 1500/3000: get.num test out this program↵
The number of arguments is: 4
Stardent 1500/3000:
```

*Special Parameter: $**

The **$*** parameter within a shell program contains all the arguments with which the shell program was invoked, starting with the first. (You are not restricted to nine parameters as with the positional parameters **$1** through **$9**.)

In the following example, the executable shell program called **show.param** contains one line.

```
Stardent 1500/3000: cat show.param ↵
echo The parameters for this command are: $*
Stardent 1500/3000:
```

show.param echoes all the arguments you give to it. For example

```
Stardent 1500/3000: show.param Hello. How are you? ↵
The parameters for this command are: Hello.  How are you?
Stardent 1500/3000: show.param one two 3 4 5 six 7 8 9 10 11 ↵
The parameters for this command are: one two 3 4 5 six 7 8 9 10 11
Stardent 1500/3000:
```

The **$*** parameter allows you to refer to multiple files using a single command line argument. For example, if you have several files in your directory named for chapters of a book: **chap1** through **chap8**, use **show.param** to print a list of all those files:

```
Stardent 1500/3000: show.param chap? ↵
The parameters for this command are: chap1 chap2
chap3 chap4 chap5 chap6 chap7 chap8
Stardent 1500/3000:
```

Named Variables

Shell programs accommodate two types of named variables:

- User-named.
- Reserved.

These are described in this section.

You can name your own shell program variables and assign values to them yourself. You do this in a shell program by adding the line:

named_variable=value ↵

Then, later in the program, you refer to the variable with a prefix: *$named_variable.*

Here is an example:

```
Stardent 1500/3000: cat hi ↵
person=Sally
echo Hello $person
Stardent 1500/3000:
```

The variable **person** is assigned the value **Sally**. The character string **$person** then refers to **Sally** (the value of the variable). Think of the **$** prefix as denoting "value". Here is the output of the executable program.

```
Stardent 1500/3000: hi ↵
Hello Sally
Stardent 1500/3000:
```

The first character of a variable must be a letter or an underscore. The rest of the name can be composed of letters, underscores, and digits. As in shell program file names, it is not advisable to use a shell command name as a variable name.

Reserved variables have specific assigned meanings and cannot be redefined by the user. A brief explanation of these follows.

- **CDPATH** defines the search path for the **cd** command.
- **HOME** is the default variable for the **cd** command (home directory).

- IFS defines the internal field separators (normally the space, tab, and carriage return).
- LOGNAME is your login name.
- MAIL names the file that contains your electronic mail.
- PATH determines the search path used by the shell to find commands and other executable programs.
- PS1 defines the primary prompt (default is $).
- PS2 defines the secondary prompt (default is >).
- TERM identifies your terminal type. It is important to set this variable if you are editing with **vi**.
- TERMINFO identifies the directory to be searched for information about your terminal, for example, its screen size.
- TZ defines the time zone (default is **EST5EDT**).

Some of these reserved variables are explained in *Modifying Your Login Environment* later in this chapter. You can also read more about them on the **sh**(1) manual page in the *Commands Reference Manual.*

You set the values of reserved variables in the same way you set user-named variables. For example, you use

TERM=*term_name* ↵

to set the value of the variable **TERM** before using the **vi** editor.

At any time you can issue the command

echo *$reserved_var* ↵

to get the current value of the variable *reserved_var*. Try typing

echo $TERM ↵

to get the name of your terminal.

To get the current values of all your reserved variables, issue the "environment" command:

env ↵

Using *variable_name=value* is an easy way to assign a value to a variable. Alternatively, do it in any of the following ways.

- Use the **read** command to assign input to the variable.
- Redirect the output of a command into a variable by using command substitution with grave accents (`...`).
- Assign a positional parameter to the variable.

The following sections discuss each of these methods in detail.

Using the read *Command*

The **read** command used within a shell program allows you to prompt the user of the program for the values of variables. The general format for the **read** command is

read *variable* ↵

The values assigned by **read** to *variable* are substituted for *$variable* wherever it is used in the program. The **read** command waits until you type in a character string followed by ↵, and then makes that string the value of the variable.

In the following example an executable shell program called **num.please** keeps track of telephone numbers listed in the file **listf1.**

```
Stardent 1500/3000: cat num.please ↵
echo Last name, please:
read name
grep $name list
Stardent 1500/3000:
```

If your *list* file contains the following names,

```
Stardent 1500/3000: cat list ↵
Wilson 408-732-0600
Smith 415-762-9888
Jones 712-984-0435
Stardent 1500/3000:
```

then you can execute **num.please** as follows.

```
Stardent 1500/3000: num.please ↵
Last name, please:
Smith ↵
Smith 415-762-9888
Stardent 1500/3000:
```

The next example is a shell program called **mknum** that creates a phone list. **mknum** includes the following commands.

- **echo** prompts for a person's name
- **read** assigns the person's name to the variable *name*
- **echo** asks for the person's number
- **read** assigns the telephone number to the variable *num*
- **echo** adds the values of the variables *name* and *num* to the file **list**

The finished program looks like this.

```
Stardent 1500/3000: cat mknum ↵
echo Type in name
read name
echo Type in number
read num
echo $name $num >> list
Stardent 1500/3000:
```

In the next example, **mknum** creates a new listing for Mr. Niceguy. **num.please** then gives you Mr. Niceguy's phone number.

```
Stardent 1500/3000: sh mknum ↵
Type in the name
Mr. Niceguy ↵
Type in the number
668-0007 ↵
Stardent 1500/3000: sh num.please ↵
Type in last name
Niceguy ↵
Mr. Niceguy 668-0007
Stardent 1500/3000:
```

Substituting Command Output for the Value of a Variable

You can substitute a command's output for the value of a variable by using *command substitution*. Use the following format in your shell program.

variable=`*command*` ↵

The output from *command* becomes the value of *variable*. The command must be enclosed in grave accents (`..`).

In the following example, the **date** command is piped into the **cut** command (to strip off the day and year portions of **date**). This produces an executable shell program **what_time** that prints the time.

```
Stardent 1500/3000: cat what_time ↵
time=`date | cut -c12-19`
echo The time is: $time
Stardent 1500/3000: what_time ↵
The time is: 10:36
Stardent 1500/3000:
```

(See **cut**(1) in the *Commands Reference Manual* for more on the **cut** command.)

Assigning Values with Positional Parameters

You can assign a positional parameter to a named parameter by using the following format:

var1=$*n* ↵

where $*n* is the *n*th argument on the command line.

This example shows an executable program that assigns positional parameters to two variables. The program is then run with command line arguments.

```
Stardent 1500/3000: cat arg_assign ↵
var1=$1
var2=$2
echo This is the first argument:  $var1
echo This is the second argument:  $var2
Stardent 1500/3000: arg_assign Hello There ↵
This is the first argument:  Hello
This is the second argument:  There
Stardent 1500/3000:
```

Shell Programming Constructs

The shell programming language has several constructs that give added flexibility to your programs.

- Comments let you document a program's function.
- Return codes signal whether a program has executed successfully.
- The looping constructs, **for** and **while**, allow a program to iterate through groups of commands in a loop.
- The conditional control commands, **if** and **case**, execute a group of commands only if a particular set of conditions is met.
- The **break** command allows a program to exit unconditionally from a loop.

Comments

You can place comments in a shell program by using the # (pound) sign. All text on a line following a # sign is ignored by the shell. The # sign can be at the beginning of a line, in which case the comment uses the entire line, or it can occur after a command, in which case the command is executed but the remainder of the line is ignored. The end of a line always ends a comment. The general format for a comment line is

#comment ↵

Consider the following program:

```
# This program sends a generic birthday greeting.
# This program needs a login as
# the positional parameter.
echo THE END  #This is the end of the program.
```

The shell ignores everything in the program except the phrase

```
echo THE END
```

Comments are useful for documenting a program's function and should be included in the programs you write.

Return Codes

Most shell commands produce return codes that indicate whether or not the command executed properly. By convention, if the value returned is 0 (zero) the command executed properly; any other value indicates that it did not. The return code is not printed automatically, but is available as the value of the shell special parameter **$?**.

After executing a command interactively, you can see its return code by typing

echo $?

Consider the following example:

```
Stardent 1500/3000: cat hi ↵
This is file hi.
Stardent 1500/3000: echo $? ↵
0
Stardent 1500/3000: cat hello ↵
cat: cannot open hello
Stardent 1500/3000: echo $? ↵
2
Stardent 1500/3000:
```

In the first case, the file *hi* exists in your directory and has read permission for you. The **cat** command behaves as expected and outputs the contents of the file. It exits with a return code of 0, which you can see using the parameter **$?**. In the second case, the file either does not exist or does not have read permission for you. The **cat** command prints a diagnostic message and exits with a return code of 2.

Return codes are useful if you want a program to execute a command conditional on whether or not a previous command was successful. See below, *Using the Test Command With Return Codes*.

Looping With the for Loop

In the previous examples in this chapter, the commands in shell programs have been executed in sequence. The **for** and **while** looping constructs allow a program to execute a command or sequence of commands several times.

The **for** loop executes a sequence of commands once for each member of a list. It has the following format:

for *variable* **in** *a_list_of_values* ↵
do ↵
    command 1 ↵
    command 2 ↵
        .
        .
        .
    last command ↵
done↵

For each iteration of the loop, the next member of the list is assigned to the variable given in the **for** clause. References to that variable may be made anywhere in the commands within the **do** clause.

As an example, the following program moves the files memos1, memos2, and memos3 to a new directory.

```
Stardent 1500/3000: cat mv.file ↵
echo Please type in the directory path
read path
for file in memo1 memo2 memo3
do
   mv $file $path/$file
   echo $file has been moved to $path/$file
done
Stardent 1500/3000:
```

The program contains these constructs:

echo
Prompts the user for a path name to the new directory.

read
Assigns the pathname to the variable **path**.

for *variable*
Calls the variable **file**; it can be referred to as **$file** in the command sequence.

in *list_of_values*
Supplies a list of values. If the **in** clause is omitted, the list of values is assumed to be **$*** (all the arguments entered on the command line).

do *command_sequence*
Provides a command sequence. In this case it is

```
do
    mv $file $path/$file
      echo $file has been moved to $path/$file
done
```

Notice several things about the program.

- Indentation is used to make the **for** and **do** loops visually clear. This is good programming style and causes no confusion because the shell ignores blanks at the beginning of lines.
- The variable can be any name you choose. If the name is *var*, just be sure you use *$var* to refer to the value of the variable later in the program.
- If **in** is omitted the shell looks for the values of the variable as command line arguments (as if the positional parameter **$*** were specified.
- The command list between **do** and **done** is executed once for each value of the variable.

In the following version of the program **in** is omitted, so the value for file is accepted as a command line argument.

```
Stardent 1500/3000: cat mv.file ↵
echo type in the directory path
read path
for file
do
   mv $file $path/$file
   echo $file has been moved to $path/$file
done
Stardent 1500/3000:
```

Try moving files with this program. You supply the name of the file you want to move as an argument on the command line. (To move several files try using the file name expansion characters (*, ?, or [] on the command line.)

Looping With the while - do Loop

Another loop construct, the **while - do** loop, allows you to do a list of tasks repeatedly based on a list of conditions. Its general format is is

while ↵
command 1 ↵
.
.
.
last command ↵
do ↵
command 1 ↵
.
.
.
last command ↵
done↵

For each iteration of the loop the following occurs. First, the set of commands in the **while** list is executed. As long as the last command in that list executes successfully (has a zero exit or return code), the shell proceeds to do all the commands in the **do** list. When the last command in the **while** list no longer executes successfully the loop ends and any commands below the **done** keyword are executed.

For example, the following program uses a **while** loop to enter a list of names into a file.

```
Stardent 1500/3000: cat enter.name ↵
echo Please type in each person's name and then RETURN
echo Please end the list of names with CONTROL d
while read x
do
   echo $x>>xfile
done
echo xfile contains the following names:
cat xfile
Stardent 1500/3000:
```

Notice that after the loop is completed, the program executes the commands below the **done**. Here are the results of **enter.name**.

```
Stardent 1500/3000:  enter.name ↵
Please type in each person's name and then a RETURN
Please end the list of names with CONTROL d
Mary Lou ↵
Janice  ↵
 CTRL d
xfile contains the following names:
Mary Lou
Janice
Stardent 1500/3000:
```

Notice that after the loop completes, the program prints all the names contained in *xfile*.

Here is another example in which the executable program uses two commands in each of the **while** and **do** lists.

```
Stardent 1500/3000: cat read.nums ↵
while
        echo Type in two numbers, separated by a space:
        read x y
do
        echo Here is the answer:
        echo The numbers are $x and $y
done
Stardent 1500/3000: read.num ↵
Type in two numbers, separated by a space:
12 30 ↵
Here is the answer:
The numbers are 12 and 30
Type in two numbers, separated by a space:
15 6 ↵
Here is the answer:
The numbers are 15 and 6:
Type in two numbers, separated by a space:
 CTRL d
Stardent 1500/3000:
```

The Shell's Garbage Can: /dev/null

The file system has a file called */dev/null* where you can have the shell deposit any unwanted output.

Try out */dev/null* by destroying the results of the **who** command. First, type in the **who** command. The response tells you who is on the system. Now, try the **who** command, but redirect the output into */dev/null*:

who > /dev/null ↵

Notice that the system responds with a prompt. The output from the **who** command was placed in **/dev/null** and was effectively discarded.

If...the Conditional Constructs

The **if** command tells the shell program to execute the **then** sequence of commands only if the command following **if** is successful. The **if** construct ends with the keyword **fi**.

The general format for the **if** construct is

if *command1* ↵
then ↵
command1 ↵
.
.
.
last command ↵
fi ↵

The following shell program called **search** demonstrates the use of the **if...then** construct. **search** uses the **grep** command to search for a word in a file. If **grep** is successful, the program will **echo** that the word is found in the file.

```
Stardent 1500/3000: cat search ↵
echo Type in the word and the file name.
read word file
if grep $word $file
   then
        echo You found it!
        echo $word is in $file
fi
Stardent 1500/3000:
```

Notice that the **read** command assigns values to two variables. The first characters you type, up until a space, are assigned to **word**. The rest of the characters, including embedded spaces, are assigned to **file**.

A problem with this program is the unwanted display of output from the **grep** command. If you want to dispose of the system response to the **grep** command in your program, use the file */dev/null*, changing the **if** command line to the following:

if grep $word $file > /dev/null ↵

if...then...else Conditional Constructs

In the event that the command in the **if** clause of the **if...then** construction is false, use an **else** clause issue an alternative set of commands.

if *command1* ↵
then ↵
command1 ↵
.
.
.
last command ↵
else ↵
command1 ↵
.
.
.
last command ↵
fi ↵

You can now improve your **search** command so it tells you when it cannot find a word, as well as when it can. The following screen shows how your improved program looks:

```
Stardent 1500/3000: cat search ↵
echo Type in the word and the file name.
read word file
if
   grep $word $file >/dev/null
then
   echo You found it!
   echo $word is in $file
else
   echo Tough luck, the word wasn't there!
   echo $word is NOT in $file
fi
Stardent 1500/3000:
```

The test *Command for Loops*

The **test** command, which checks to see if certain conditions are true, is a useful command for conditional constructs. If the condition is true, the loop continues. If the condition is false, the loop ends and the next command is executed. Some of the useful options for the **test** command are

test **–r** *file*
True if the file exists and is readable.

test **–w** *file*
True if the file exists and has write permission.

test **–x** *file*
True if the file exists and is executable.

test **–s** *file*
True if the file exists and has at least one character.

test *var1* **–eq** *var2*
True if *var1* equals *var2*.

test *var1* **–ne** *var2*
True if *var1* does not equal *var2*.

The following executable shell program named **mv.ex** moves all the executable files in your current directory to your *bin* directory. This program uses the **test –x** command in the **do...done** loop to select the executable files. The variable **$HOME** gives the path to the login directory. **$HOME/bin** gives the path to your *bin*.

```
Stardent 1500/3000: cat mv.ex ↵
for file
  do
    if test -x $file
       then
         mv $file $HOME/bin/$file
    fi
  done
Stardent 1500/3000:
```

The next example shows you how to test the **mv.ex** command using all the files in the current directory, specified with the * metacharacter as the command argument. The command lines shown in this example execute the **mv.ex** from the current directory and then change to *bin* and list the files in that directory. All executable files should be in *bin*.

```
Stardent 1500/3000: mv.ex * ↵
Stardent 1500/3000: cd; cd bin; ls ↵
list_of_executable_files
Stardent 1500/3000:
```

Using the Test Command With Return Codes

The test command can be used along with return codes to permit you to execute a command only if a previous command executes successfully. For example, the following program reads in a word and a file and sends the file to the line printer only if the word is in the file.

```
Stardent 1500/3000: cat maybe.print ↵
echo Please type in the word and file name, separated by a space:
read word file
grep $word $file > /dev/null
if test "$?" -eq "0"
       then
           echo The word was found, the file will be printed.
           lpr $file
       else
           echo Sorry, the word is not in the file.
fi
Stardent 1500/3000:
```

case..esac Conditional Constructs

The **case...esac** construction has a multiple choice format that allows you to choose one of several patterns and then execute a list of commands for that pattern. The pattern statements must begin with the keyword **in**, and a) must be placed after the last character of each pattern. The command sequence for each pattern is ended with ;;. The **case** construction must be ended with **esac** (the letters of the word case reversed).

The general format for the **case** construction is

case *word* ↵
in ↵
pattern1) ↵
command line 1 ↵
.
.
.
last command line ↵
;; ↵

```
pattern2)↵
                    command line 1↵
        .
        .
        .
   last command line↵
        ;;↵
pattern3)↵
                    command line 1↵
        .
        .
        .
   last command line↵
        ;;↵
*)↵
                    command 1↵
        .
        .
        .
   last command↵
        ;;↵
esac↵
```

The **case** construction tries to match the *word* following the keyword **case** with the *pattern* in the first pattern section. If there is a match, the program executes the command lines after the first pattern and up to the corresponding ;;.

If the first pattern is not matched, the program proceeds to the second pattern. Once a pattern is matched, the program does not try to match any more of the patterns, but goes to the command following **esac**.

In the following example the **set.term** shell program contains an example of the **case...esac** construction. This program sets the shell variable TERM according to the type of terminal you are using. In this example, the terminal is a Teletype 4420, Teletype 5410, or Teletype 5420.

set.term first checks to see whether the value of **term** is 4420. If it is, the program makes T4 the value of **TERM**, and terminates. If the value of **term** is not 4420, the program checks for other values: 5410 and 5420. It executes the command under the first pattern it finds, and then goes to the first command after **esac**.

Notice the use of * as the last pattern in the **case** statement. The special character * matches any character string and so allows you

to give a set of commands to be executed if no other pattern matches. To do this, it must be placed as the last possible pattern in the **case** construct, so that the other patterns are checked first. This provides a useful way to detect erroneous or unexpected input.

(Any of the metacharacters *, ?, and [ ] can be used as part of a pattern in a **case** statement. This permits the use of the file name expansion for added flexibility.)

Here the * pattern is used to warn that you do not have a pattern for the terminal specified and allows you to exit the **case** construct.

```
Stardent 1500/3000: cat set.term ↵
echo If you have a TTY 4420 type in 4420
echo If you have a TTY 5410 type in 5410
echo If you have a TTY 5420 type in 5420
read term
case $term
        in
                4420)
                        TERM=T4
                ;;
                5410)
                        TERM=T5
                ;;
                5420)
                        TERM=T7
                ;;
                *)
                echo not a correct terminal type
                ;;
esac
export TERM
echo end of program
Stardent 1500/3000:
```

Notice the use of the **export** command. You use **export** to make a variable available within your environment and to other shell procedures.

Unconditional Control Statements: the break *and* continue *Commands*

The **break** command unconditionally stops the execution of any loop in which it is encountered, and goes to the next command after the **done, fi,** or **esac** statement. If there are no commands after that statement, the program ends.

In the example for **set.term**, you could have used the **break** command instead of **echo** to leave the program, as the next example shows:

```
Stardent 1500/3000: cat set.term ↵
echo If you have a TTY 4420 type in 4420
echo If you have a TTY 5410 type in 5410
echo If you have a TTY 5420 type in 5420
read term
case $term
        in
                4420)
                TERM=T4
        ;;
        5410)
                TERM=T5
        ;;
        5420)
                TERM=T7
        ;;
        *)
                break
        ;;
esac
export TERM
echo end of program
Stardent 1500/3000:
```

The **continue** command causes the program to go immediately to the next iteration of a loop without executing the remaining commands in the loop. It can be used to simplify complicated constructs such as **if - then - else** statements nested within **while - do** or **for** loops. It is less commonly used than the other constructs discussed here.

Debugging Programs

At times you may need to debug a program to find and correct errors. There are two options to the **sh** command (listed below) that can help you debug a program:

sh –v *shellprogramname*
Prints the shell input lines as they are read by the system.

sh –x *shellprogramname*
Prints commands and their arguments as they are executed.

Here is an example to show how debugging works. The executable program **in.dir** uses the command **basename,** which strips off any prefix ending in / from a file or directory name. **basename** works as follows:

```
Stardent 1500/3000: basename /usr/starship/draft/outline ↵
outline
Stardent 1500/3000:
```

in.dir accepts as input the full pathname of a file. It checks to see if the file is in the current directory. If yes, a message to that effect is printed. If not, a different message is printed.

```
Stardent 1500/3000: cat in.dir ↵
ls `basename $1` > /dev/null
if test "$?" -eq "0"
then
        echo The file is in the directory!
else
        echo Sorry, the file is not in the directory!
fi
```

The program uses command substitution on the first line, substituting the output of the **basename** command into the **ls** command. Notice the grave accents (`..`) that are required for command substitution to work. It then uses the return code to determine whether or not the **ls** command was successful, and prints an appropriate message. For example, suppose you are in the directory */usr/starship/draft*, which contains the file *outline*. Run the program as follows:

```
Stardent 1500/3000: in.dir /usr/starship/draft/outline ↵
The file is in the directory!
Stardent 1500/3000:
```

Now suppose you wrote this program, but instead of using grave accents around the **basename** command you used single quotes. You would get the following incorrect output:

```
Stardent 1500/3000: in.dir /usr/starship/draft/outline ↵
Sorry, the file is not in the directory!
Stardent 1500/3000:
```

To help find the mistake, issue the following debugging command:

sh -x in.dir /usr/starship/draft ↵

Here is the result:

```
Stardent 1500/3000: sh -x in.dir /usr/starship/draft/outline ↵
+ ls basename $1
basename $1 not found
+ test 2 = 0
+ echo Sorry, the file is not in the directory!
Sorry, the file is not in the directory!
Stardent 1500/3000:
```

(The lines with the prefix + represent lines in the program. They are printed as they are executed. The other lines represent actual output of the program.)

You can see the problem with this output. The **ls** command is not being run on the actual file name. Instead, the shell believes that the string "basename $1" is the name of the file. Consequently the test fails, and the wrong answer is given.

Compare the output of the debugging command when the grave accents are restored to the program:

```
Stardent 1500/3000: sh -x in.dir /usr/starship/draft/outline ↵
+ basename /usr/starship/draft/outline
+ ls outline
+ test 0 = 0
+ echo The file is in the directory!
The file is in the directory!
```

You can see that the program takes the basename of the right file, and that the test is executed correctly.

To debug pipelines use the **tee** command. While simply passing its standard input to its standard output, it also saves a copy of its input into the file whose name is given as an argument.

The general format of the **tee** command is

command1 | **tee** *saverfile* | *command2* ↵

saverfile is the file that saves the output of *command1* for you to study.

For example, say you want to check on the output of the **grep** command in the following command line:

who | grep $1 | cut –c1-9 ↵

You can use **tee** to copy the output of **grep** into a file called *check*, without disturbing the rest of the pipeline.

who | grep $1 | tee check | cut –c1-9 ↵

The file *check* contains a copy of the **grep** output, as shown in the following screen:

```
Stardent 1500/3000: who | grep mlhmo | tee check | cut –c1-9 ↵
mlhmo
Stardent 1500/3000: cat check ↵
mlhmo   tty61   Apr 10   11:30
Stardent 1500/3000:
```

The C-Shell

The C-shell is an enhanced version of the shell that can be used instead of the shell as a command interface and programming language. Its most useful capabilities include

- A history feature that gives you shorthand ways of repeating or altering previously issued commands.
- An alias feature that gives you a simple way of modifying or redefining commands.
- A job control feature to move processes back and forth between the foreground and background, and to start or stop processes.

The C-shell also has programming capabilities not available in the shell, and the ability to define numeric and array variables.

A description of the C-shell can be found in the *Commands Reference Manual.* (Also, see the *Preface* to this guide for information about books that contain more about the C-shell.) This section

describes the history and alias features.

Using the C-Shell

Your environment may already be set up so that you use the C-shell whenever you log in, or you can choose to invoke the C-shell in a particular session. You may also assign individual windows to run the shell or the C-shell.

To find out whether you are currently running the C-shell, issue the **ps** command:

```
Stardent 1500/3000: ps ↵
   PID TTY          TIME COMMAND
   218 sxt002       0:00 csh
   271 sxt002       0:01 ps
Stardent 1500/3000:
```

In this example the C-shell is running, as indicated by "csh" as the first command. If the shell were running, "sh" would have been printed instead. To move from the shell to the C-shell you use the command

csh ↵

To move from the C-shell to the shell you type

sh ↵

To set up your environment to run the C-shell, a file called */etc/passwd/* must be edited. (This should be done by your system administrator. If you are responsible for your own system administration, see *The /etc/passwd File* in the *System Administrator's Guide.*)

The C-shell programming language includes many more capabilities than the shell; however, any shell programs can be run in the C-shell environment (as long as the code for the program resides in a file). Files containing C-shell programs must have a # as their first character.

The C-Shell History Feature

The C-shell history feature allows you to return to previously issued commands, to reissue, modify, or simply remember them. You use the **history** command to get a list of recently issued commands:

```
Stardent 1500/3000: history ↵
    1  mail
    2  ls
    3  cd /usr/rocket/work/cpgms
    4  ls -l
    5  vi eval.c
    6  cd /usr/bjm
    7  history
```

As the example shows commands are listed by number with the most recent commands listed last. The number of commands listed is determined by the *history* variable. (See *Modifying Your Login Environment* below.)

You have various options for reissuing a command. In each case you begin with an exclamation mark (!) on the command line. Typing just two exclamation marks repeats the last command issued. Here are two of the most useful options.

- Using the number of the command. For instance, typing **!3** ↵ repeats the **cd** command in the above example.

- Using the command name, which may be truncated. For instance, typing **!v** ↵ repeats the **vi** command in the above example. Typing **!l** ↵ repeats the most recently issued command that matches the pattern (command 4 in the example.)

To alter a previously issued command you use the substitution capability. For instance, suppose you decide you want to move to directory */usr/rocket/work/fpgms*. Issue the following command:

!3:s/cpgms/fpgms/ ↵

The **3** refers to command 3 (in the above example), the colon (:) is a delimiter, and the remainder of the command shows you are substituting the directory *fpgms* for *cpgms*. The C-shell uses line editor commands for altering previously issued commands. See **ed**(1) in the *Commands Reference Manual* for a description of line editor commands.

The C-Shell Alias Feature

The C-shell gives you a simple way of redefining commands. This is useful if you want to shorten some frequently-used but lengthy commands, or if you always use certain options and don't want to have to list them explicitly. You use the following format:

alias *entered_command executed_command* ↵

Here are a couple of examples.

```
Stardent 1500/3000: alias ls ls -l ↵
Stardent 1500/3000; alias cdprog cd /usr/rocket/work ↵
Stardent 1500/3000:
```

The first example shows the **ls** command redefined so that the long option, **–l** is invoked every time. The second example gives a shorthand command, **cdprog**, for moving to a particular directory. Here is the result of defining the command **cdprog**:

```
Stardent 1500/3000: cdprog ↵
Stardent 1500/3000: pwd ↵
/usr/rocket/work
Stardent 1500/3000:
```

To get a list of current aliases, type

alias ↵

Modifying Your Login Environment

Every time you log in to your system the shell executes one or more special "dot" files in your home directory. These files control your login environment and can be customized as you wish. If you are using the standard shell, a file named *.profile* (pronounced "dot profile") is executed; if you are using the C-shell, two files, *.login* and *.cshrc* are executed. The *.profile* or *.login* and *.cshrc* files allow you to set the values of shell variables such as TERM and PATH and to set terminal options such as BACKSPACE. (See *Named Variables* in this chapter for more on shell variables.)

It is a good idea to customize all three files (*.profile, .login, .cshrc*) if you want the option of using either shell interface. Procedures for modifying these files are described below.

Modifying Your .profile

Because the *.profile* is a file, it can be edited and changed to suit your needs. You may already have made some of the changes described earlier in this guide, for instance, adding the name of your terminal so use the **vi** editor.

Before making any changes to your *.profile*, make a copy of it in another file called *safe.profile*. This ensures that you can recover your current *.profile* in the event you don't like the changes you have made. Type

cp .profile safe.profile ↵

You can now add commands to your *.profile* just as you do with any other shell program. Practice adding commands to your *.profile*. Edit the file and add the following **echo** command to the last line of the file:

echo Hello! Let's Go!

Whenever you want to implement changes to your *.profile* during the current work session you may do so by using the . (dot) shell command. This executes the commands in your *.profile* thus reinitializing your environment: Try this now. Type

. .profile ↵

The system should respond with the following:

```
Hello! Let's Go!
```

An Example .profile

Here is an example *.profile* that contains terminal settings and reserved variable assignments.

```
Stardent 1500/3000: cat .profile ↵
stty -tabs
stty echoe
PATH=$PATH:$HOME/bin:/project/lib
TERM=vt100
EDITOR=vi
PS1=<>
export PATH TERM PS1 EDITOR
# a message:
echo Hello! Let's go!
```

(The line beginning with # is a comment.) The next sections describe some of the modifications you may want to make to your *.profile*.

Setting Terminal Options

The **stty** command can make your shell environment more convenient. Three commonly used options are **–tabs, erase** CTRL h, **and echoe.**

stty –tabs
This option preserves tabs when you are printing. It expands the tab setting to eight spaces, which is the default. (See **stty**(1) in the *Commands Reference Manual* for more on the **tabs** options.)

stty erase CTRL h
This option allows you to use the erase key on your keyboard to erase a letter, instead of the default character #. Usually ⌫ is the erase key.

stty echoe
This option erases characters from the screen as you erase them with ⌫.

If you want to use these options for the **stty** command, just add the appropriate command lines to your *.profile* as in the above example.

Using Shell Variables

The values of reserved shell variables can be set in your *.profile*. (See *Named Variables* in this chapter for a full list of reserved variables.) First, to see what variables have already been set, issue the **env** (environment) command:

env ↵

As the following example (for login *rocket*) shows, you may have numerous variables already set, either in your *.profile*, or in other files used in system administration.

```
Stardent 1500/3000: env ↵
LOGNAME=rocket
HOME=/usr/rocket
PATH=$PATH:$HOME/bin:/usr/bin:/usr/lib
CDPATH=.:$HOME
TERM=vt100
MAIL=/usr/mail/rocket
TZ=PST8PDT
PS1=Stardent 1500/3000:
PS2=>
Stardent 1500/3000:
```

If you are interested in the value of a particular shell variable, you can issue the following command:

echo $*variable_name* ↵

If you wish to set the values of any variables not yet defined, or if a variable is defined incorrectly, make the additions or changes in your *.profile*. To do so you must both add an assignment line of the form

VARIABLE=value

and make sure that the variable is "exported", by adding the line

export VARIABLE

to the *.profile*. (If your *.profile* already has an **export** line you can simply add the new variable to it. See the example *.profile* above.)

Here are a few of the variables you may want to set.

PATH

This variable gives the search path for finding and executing commands. Notice the line in the example *.profile* above:

PATH=$PATH:$HOME/bin:/project/lib

This line, defining the value of the variable PATH, gives a list of pathnames where the shell should search for executable programs. The colons (:) are delimiters between the pathnames assigned to the variable PATH.

The first element in the list, $PATH, includes pathnames to executable programs that you get automatically as a user of the system (pathnames such as */usr/bin* and */etc/passwd*). It is necessary to include $PATH in the list since you are redefining the value of the variable PATH.

The next pathname, $HOME/*bin*, gives the path to the bin directory in the user's home directory. The last pathname, */project/lib*, gives the path to the directory */project/lib*, a library of executable programs for the user's project.

If you wish to add a pathname to a list such as this, simply add a colon (:) and the desired pathname to the end of the line.

TERM

This variable tells the shell what kind of terminal you are using. Add a line of the form

TERM=*terminal_name*

to your *.profile*. (See the example *.profile* above.) It is necessary to assign the value of TERM if you plan to use a screen editor such as **vi**.

NOTE
Don't forget to export all the variables you set in your *.profile*!

EDITOR

This variable tells the system what screen editor you are using. (It is required by some system programs.) To indicate that you are using the **vi** editor, add the line

EDITOR=vi

to your *.profile*.

PS1

This variable sets the primary shell prompt string (the default is Stardent 1500/3000:).

Try the following example. Note that to use a multi-word prompt, you must enclose the phrase in quotes. Type the following variable assignment in your *.profile*, and export the variable PS1. Then execute your **.profile** (with the . command) and watch for your new prompt sign.

```
Stardent 1500/3000: ..profile ↵
Your command is my wish:
```

You can also put any messages or commands you wish to run automatically when you log in, in your *.profile*. For instance, if you add the command **who** to the file, the system will tell you who is currently logged on every time you log in.

Modifying Your C-Shell Environment: The .login

and *.cshrc* Files" Modifying your C-shell environment is analogous to modifying your shell login environment, but with some important differences. (Because of the similarities, be sure to read *Modifying Your .profile* above before reading this section.)

- Many of the C-shell variable names are different. (For instance, you use path (lower case) instead of PATH to define the search path for commands and prompt instead of PS1 to set the system prompt. See **csh**(1) in the *Commands Reference Manual* for a complete list of C-shell variables.) You can, however, set the values of some standard shell variables.

- The syntax for setting the values of variables is different.

- You use two files: *.login* and *.cshrc*.

The *.login* file is executed once, when you log in (just like the *.profile*). Like the *.profile* it should be used for terminal settings and setting the values of environment (or global) variables such as term and path.

The *.cshrc* file is executed every time a new C-shell is invoked. This happens at various times, including when you log in, when you open a new window and run the C-shell, when you change from the shell to the C-shell (using the **csh** command, or when you execute a program written in the C-shell programming language. The *.cshrc* file should be used for setting C-shell variables such as history and commands such as **alias**, which by their nature only retain their values within a given shell.

NOTE
There is no firm rule about should be included in the *.login* file versus the *.cshrc* file; in fact, various people have differing philosophies. Use the explanation here as a general guideline, not as the last word.

To set terminal options in your *.login* file use the same command you used in your *.profile*. (See *Setting Terminal Options* above.) The only difference is that the C-shell allows you to define several options on one line (see the example below).

To assign the values of environment variables use the **setenv** command:

setenv variable value ↵

To assign the values of C-shell variables use the **set** command:

set *variable* = *value* ↵

No "export" line is needed in either case.

Here are examples of a *.login* and a *.cshrc* file.

```
Stardent 1500/3000: cat .login ↵
stty erase echoe
setenv path (. /usr/bin $home/bin)
setenv term vt100
echo Hello, Let's Go!


Stardent 1500/3000: cat .cshrc ↵
set history = 10
set noclobber
set prompt = '<<>>'
alias ls ls -l
alias cdgrades cd $home/students/grades
Stardent 1500/3000:
```

C. Resources

For more information on UNIX commands, see the *Commands Reference Manual* (#340-0103-00) and the two volumes of the *Programmer's Reference Manual* (#340-0121-00 and #340-0122-00).

The command **man man** will give you information about the availability of on-line command information.

General Literature

S.R. Bourne: *The UNIX System*. Addison-Wesley, 1982.

Kaare Christian: *The UNIX Operating System*. John Wiley & Sons, 1983.

Fiedler & Hunter: *UNIX System Administration*. Hayden Books, 1986.

Kerninghan & Pike: *The UNIX Programming Environment*. Prentice-Hall, 1984.

McGilton & Morgan: *Introducing the UNIX System*. McGraw-Hill, 1983.

Prata & Martin: *UNIX System V Bible*. Howard W. Sams & Co., Inc., 1987.

Mark G. Sobell: *A Practical Guide to the UNIX System.* Benjamin/Cummings, 1984.

Thomas & Yates: *A User Guide to the UNIX System.* OSBORNE/McGraw-Hill, 1982.

Waite, Martin, Prata: *UNIX Primer Plus.* Howard W. Sams & Co., Inc., 1983.

D. Quick Reference

Operation	Command
change directory	**cd** *directoryname*
change existing permissions	**chmod** *who+permission file(s)*
copy file	**cp** *file1 file2*
count lines, words, characters	**wc** *option(s) filename*
determine existing permissions	**ls -l**
differences between files	**diff** *file1 file2*
execute commands in succession	*command1; command2; command3*
list all filenames	**ls -a**
list directory contents	**ls**
list files in long format	**ls -l**
list files in short format	**ls -CF**
make new directory	**mkdir** *directoryname*
merge files	**cat** *file1* <return> **sort** *file1 file2*
move file	**mv** *file1 file2*
peruse file contents	**pg** *filename*
print hardcopy	**lp** *option(s) filename*
print file on screen	**cat** *directoryname*
print partially formatted file	**pr** *filename*
print working directory	**pwd**
remove empty directory	**rmdir** *directoryname*
remove file	**rm** *file(s)*
rename file	**mv** *file1 file2*
return to home directory	**cd**
search file for pattern	**grep** *pattern file(s)*
sort file	**sort** *file(s)*

THE *vi* EDITOR

CHAPTER FOUR

Contents

- *A. Session One*: Commands you need to start using the text editor.
- *B. Basics*: To get you up and away for most writing and editing tasks you may have.
- *C. Resources*: Where to find more information about *vi*.
- *D. Quick Ref*: A list of the most important editing commands.

Introduction

This chapter is a guide to the visual screen editor, covering many of the editor commands for creating and editing files. It guides you through the following tasks:

- Set up the shell environment for using *vi*.
- Enter *vi*, create text, write the text to a file, and quit.
- Move text within a file.
- Cut and paste text.
- "Escape" (return) to the shell temporarily to execute shell commands.
- Use line editing commands within *vi*.
- Edit several files in the same session.
- Recover a file lost by an interrupt to an editing session.

For a complete reference to *vi* commands see the *vi*(1) pages in the *Commands Reference Manual.*

A. Session One

Start vi

vi filename

Start writing

i (to insert)

Stop writing

Esc key to change to command mode

Stop writing and save file

Esc and ZZ (note capital Z)

B. Basics

The *vi* editor functions in three different modes: command, input, and last line mode:

- In command mode, keystrokes result in issuing commands to add, delete and rearrange text.
- In input mode, you add text to the file.
- Last line mode is for line editor and pattern search commands.

vi provides a window for the file you are editing. To see and edit different parts of the file, scroll backward and forward within it. Figure 4-1 shows the view into the file you are editing.

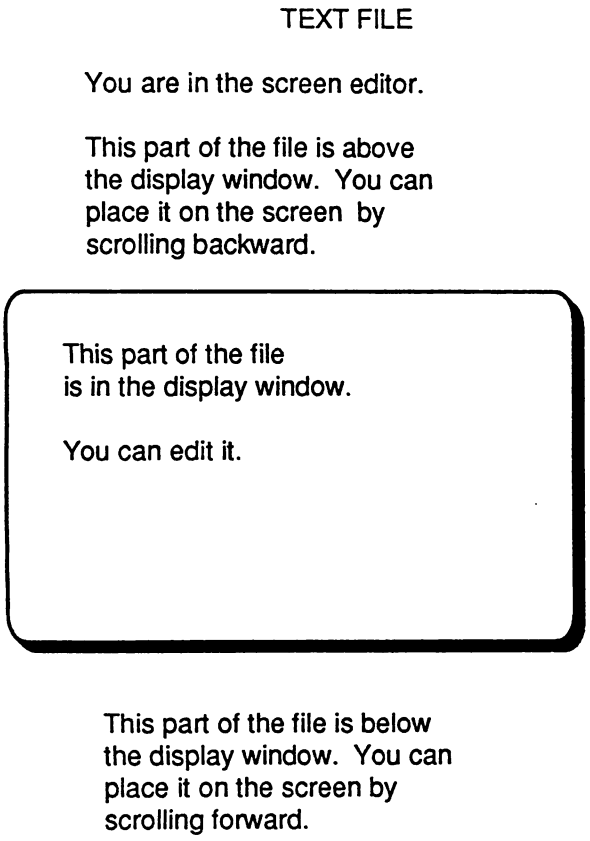


Figure 4-1. Displaying a File with a vi Window

This section is about starting *vi*. It introduces a few basic commands, most of which are described more fully later in this chapter.

Terminal Name

Before use *vi* you must provide the system with the name or type of the terminal you are using. Normally this information is contained in your *.profile*, a dot file in your home directory that is automatically executed every time you log in. (If the C-shell command interpreter is used, add your terminal type to your *.login* file. See *Modifying Your C-Shell Environment* in Chapter 3.)

Use the *pg* command to see if your *.profile* contains these two lines:

```
TERM=terminal_name
export TERM
```

If these lines are not in your *.profile* file, add them by typing at the system prompt:

```
Stardent 1500/3000: TERM=terminal_name↵
Stardent 1500/3000: export TERM↵
Stardent 1500/3000:
```

If you plan to use *vi* regularly, be sure to add these lines to *.profile;* otherwise you must type them every time you log in. See *Modifying Your Login Environment* in Chapter 3 for more on the *.profile* file.

You can always check to see if the terminal name has been set, whether or not you are currently in your home directory, by typing

echo $TERM ↵

If your terminal type has been set the system shows it; if not, you will see just the system prompt.

```
Stardent 1500/3000: echo $TERM↵
vt100
```

Creating a File

To use *vi* to create a file, type

vi *filename* ↵

where *filename* is the name of the file about to be created. (To edit an existing file, use the same procedure.) For example, to create a file named *stuff*, type

vi stuff ↵

When you type the *vi* command with the filename *stuff*, *vi* clears the screen and displays a window in which you can enter and edit text.

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

The _ (underscore) on the top line shows the cursor waiting for you to enter a command there. Every other line is marked with a ~ (tilde), the symbol for an empty line.

If, before entering *vi*, you did not set your terminal configuration as explained in *Getting Started*, above, you see an error message:

```
Stardent 1500/3000: vi stuff↵
terminal_name: unknown terminal type

[Using open mode]
"stuff" [New file]
```

You cannot set the terminal configuration while you are in *vi*; it must be done in the shell, at the system prompt. Leave the editor by typing

:q ↵

Now set the correct terminal configuration.

If *vi* doesn't allow you to return to the shell, check the error message and chances are it will tell you what to do. For example, there may be a message that you should use **q!** instead of just **q**. Follow the instructions on the screen.

Creating Text: The Input Mode

Once you have successfully entered *vi* and opened a file, you are in command mode. To create text:

- Give the *insert* command by pressing i. This puts you in input mode so that you may type text into the file.

- Type in several lines of text. Make the text lines as long or as short as you wish.
- Press the ↵ key after each line.

Leaving Input Mode

When you finish creating text press ESC to leave input mode and return to command mode. Here is an example:

i Create some text ↵
in the screen editor ↵
and return to ↵
command mode. ESC

If you press ESC and a bell sounds, you are already in command mode. The text in the file is not affected by this, even if you press ESC several times.

Editing Text: the Command Mode

The *vi* editor offers an array of commands to enable you to move within a file.

Basic Cursor Movement Commands

The following commands move the cursor around the screen:

n j	Moves the cursor down *n* lines. Without an *n* number specified, the cursor is moved down one line.
n k	Moves the cursor up *n* lines. If the *n* is omitted, the cursor is moved up one line.
n h	Moves the cursor *n* characters to the left. If you don't specify a number, the cursor is moved one character to the left.
n l	Moves the cursor *n* characters to the right. If the *n* is omitted, the cursor is moved one character to the right.

Like most *vi* commands, the j, k, h, and l commands are silent, meaning that they do not appear on the screen when you enter them. The only time you should see characters on the

screen is when you are in input mode and are adding text to your file, or when you are in last line mode, in which case characters only appear on the last line of your screen. If the cursor movement letters appear on the screen you are still in input mode. Press the ESC key to return to command mode and try the commands again.

The j and k commands maintain the column position of the cursor. For example, if the cursor is on the seventh character from the left, it goes to the seventh character on the new line when you type j or k. If there is no seventh character on the new line the cursor moves to the last character.

You can also use the arrows on your keyboard's number pad to move the cursor. Before doing so just make sure the number lock indicator is off. The number pad arrow keys correspond to the cursor movement keys discussed above: the ↓ key is equivalent to j, the ↑ key to j, the ← key to h, and the → key to l.

In addition to h and l, SPACE (space bar) and ⌫ (backspace) can be used to move the cursor right or left to a character on the current line.

n SPACE	Moves the cursor *n* characters to the right. If the *n* is omitted the cursor is moved one character to the right.
n ⌫	Moves the cursor *n* characters to the left. If the *n* is omitted the cursor is moved one character to the left.

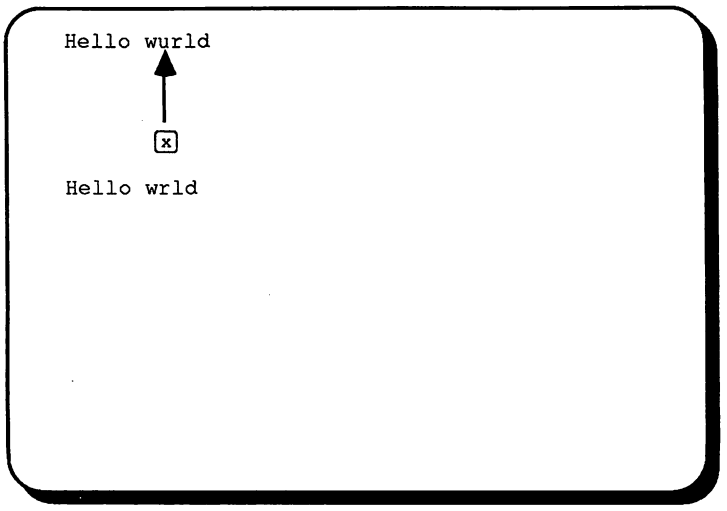
If you cannot go any farther, *vi* sounds a bell. This is true for all cursor movement commands.

Deleting Text

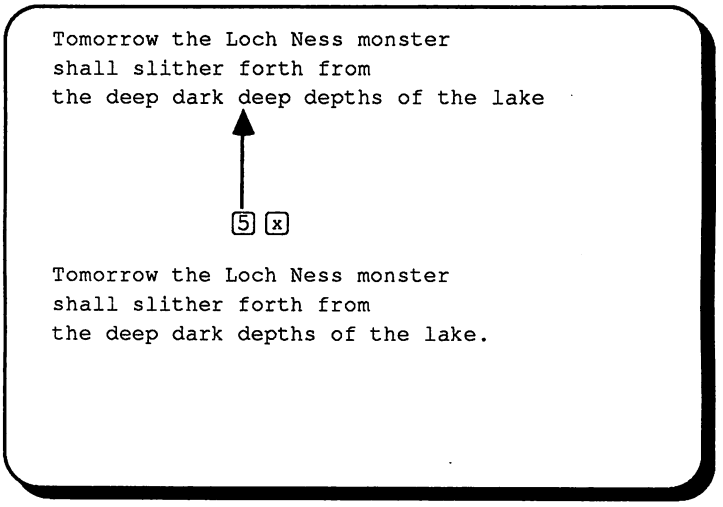
If you want to delete a character, press x. Pressing x in the command mode deletes the character under the cursor and the line readjusts to the change. In general,

n x	Deletes *n* characters, where *n* is the number of characters you want to delete. If no number is given, only one character is deleted.

In the following examples the arrows under the letters show the positions of the cursor.



To delete the second occurrence of the word *deep* from the text shown in the following screen, put the cursor on the first letter of the string you want to delete and delete five characters (for the four letters of *deep* plus an extra space).

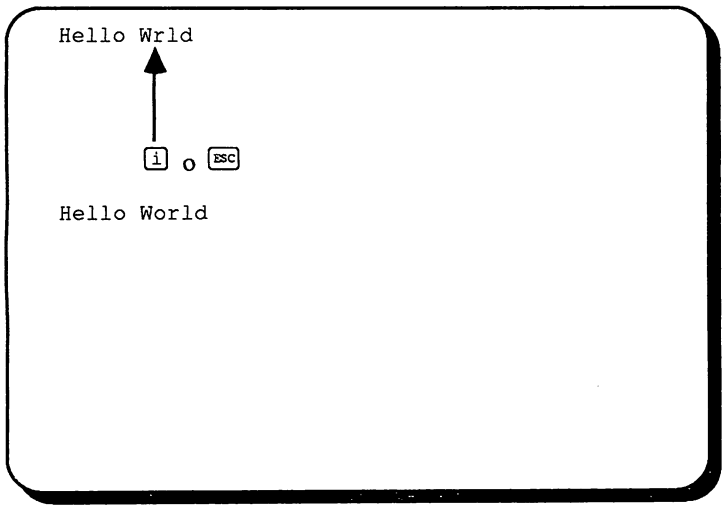


Notice that *vi* adjusts the text so that no gap appears in place of the deleted string.

Adding Text

To add text use the i (insert) or a (append) commands. To add text with the insert command, move the cursor to the place where you want to insert text and press i and start entering text. As you type, the new text appears on the screen to the left of the

cursor. The character under the cursor and all characters to the right of the cursor move right to make room for the new text. The *vi* editor continues to accept characters you type (including ↵) until you press ESC. If necessary, the original characters even wrap around onto the next line.



Use the *append* command in the same way. New text appears to the right of the cursor, not on the left as is the case with *insert*.

Quitting vi

When you enter *vi*, a copy of the file you want to edit is placed in a buffer. During your editing session all changes are made to the copy of the file, not to the original. To save your work you must make an explicit request.

To save the buffer contents in a file and return to the shell, hold down SHIFT and press Z twice (Z Z). The editor remembers the file name you specified with the *vi* command at the beginning of the editing session and moves the buffer text to the file of that name. A notice at the bottom of the screen gives the file name and the number of lines and characters in the file. The shell then gives you a prompt.

```
aThis is a test file.↵
I am adding text to↵
a temporary buffer and↵
now it is perfect.↵
I want to write this file,↵
and return to the shell. ESC Z Z
~
~
~
~
"stuff" [New file] 7 lines, 151 characters
Stardent 1500/3000:
```

You can also use the **:w** and **:q** commands of the line editor to write a file and quite the editor. Using them places you in last line mode. You see the commands appear on the last line of your screen as you type them; then, as with many of the last line commands, the command disappears when you press ↵.

:w Writes the buffer to a file.

:q Leaves the editor and returns you to the shell.

:wq Writes the buffer to a file and returns you to the shell.

```
aThis is a test file.↵
I am adding text to↵
a temporary buffer and↵
now it is perfect.↵
I want to write this file,↵
and return to the shell.↵
~
~
~
~
~
:wq↵
Stardent 1500/3000:
```

Moving the Cursor Around the Screen

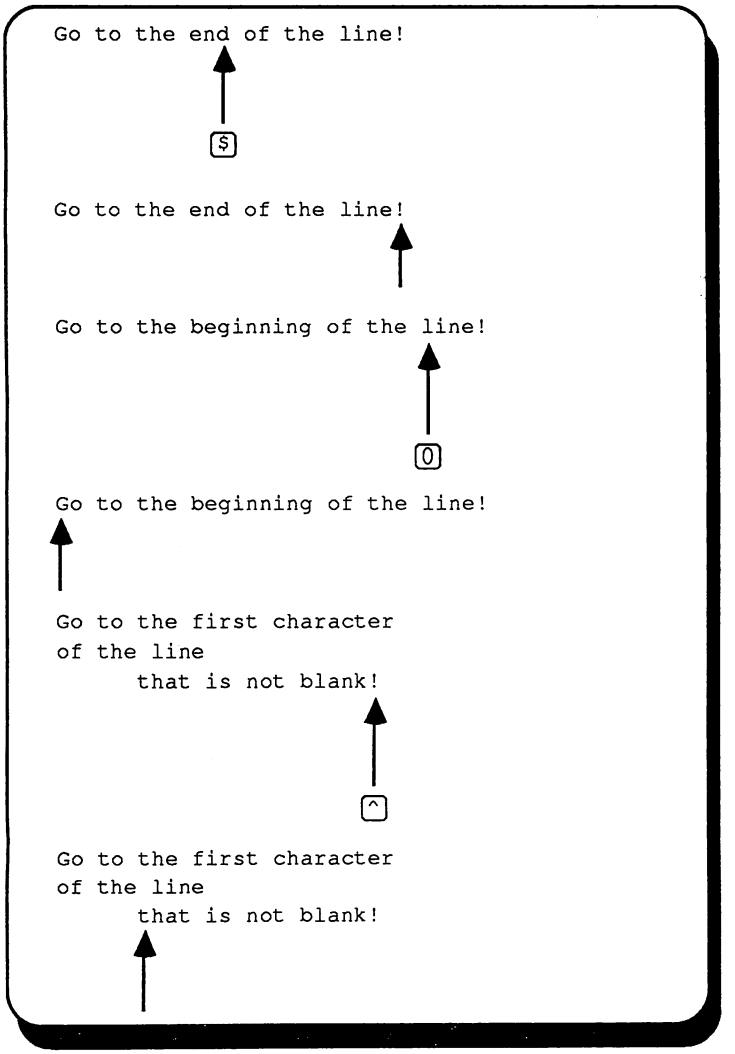
In addition to j, k, h, l, ▒, and SPACE, various other commands can help you move the cursor around the screen quickly. The following paragraphs explain how to move the cursor by characters on a line, by lines, by words, by sentences, by paragraphs, and within the window.

Moving the Cursor to the First or Last Character of a Line

To move to the beginning or end of a line use the following commands.

[$]	Puts the cursor on the last character of a line.
[0] (zero)	Puts the cursor on the first character of a line.
[^] (circumflex)	Puts the cursor on the first nonblank character of a line.

The following examples show the movement of the cursor produced by each of these three commands.

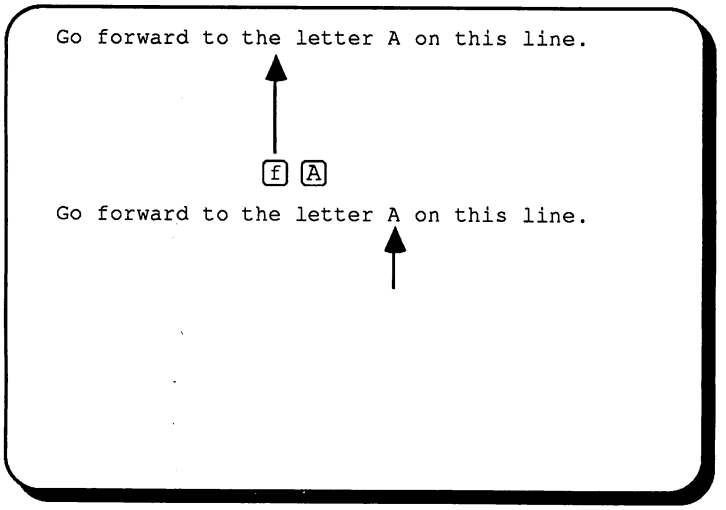


Move the Cursor to a Specific Character on a Line

The following commands allow you to search for a specific character within a line.

f *x*	Moves the cursor to the right to the specified character *x*.
F *x*	Moves the cursor to the left to the specified character *x*.
t *x*	Moves the cursor right to the character just before the specified character *x*.
T *x*	Moves the cursor left to the character just after the specified character *x*.
;	Continues the search specified in the last command, in the same direction.
,	Continues the search specified in the last command, in the opposite direction.

For example, in the following screen *vi* searches to the right for the first occurrence of the letter A on the current line.



Moving the Cursor Line by Line

In addition to the commands j and k described above, use the commands −, +, and ↵ to move the cursor to other lines.

n [-] Moves the cursor up *n* lines.

n [↵] or *n* [+]
Moves the cursor down *n* lines.

For each of these commands, if the *n* is omitted the cursor is moved one line. If there are too few lines to move in the desired direction, the cursor remains on the current line and a bell sounds.

Moving the Cursor Word by Word

The *vi* editor considers a word to be a string of characters that may include letters, numbers, or underscores. There are six word positioning commands: [w], [b], [e], [W], [B], and [E].

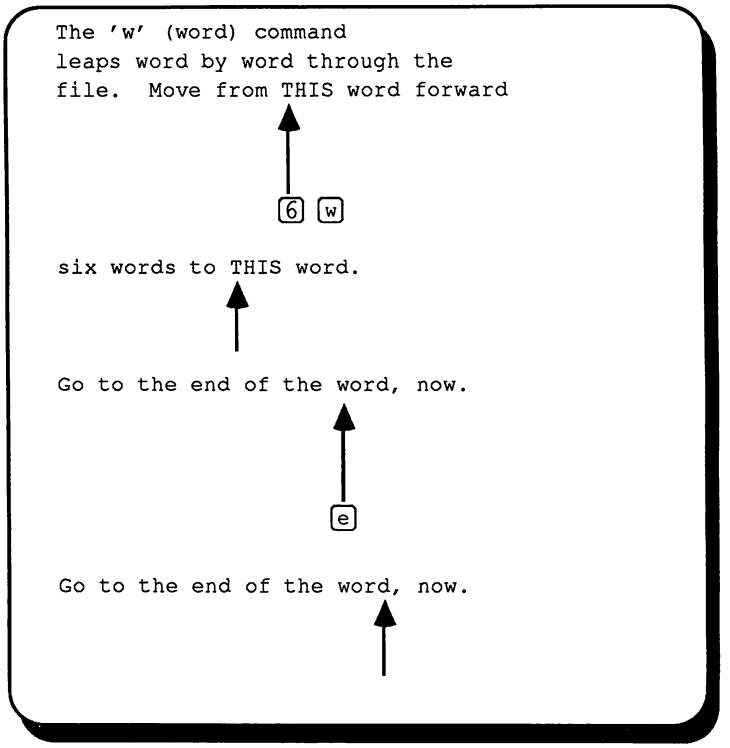
The lower case commands treat any character other than a letter, digit, or underscore as a delimiter, signifying the beginning or end of a word. The beginning or end of a line is also a delimiter. The upper case commands treat punctuation as part of the word; words are delimited only by blanks, tabs, and newlines.

The following is a summary of the word positioning commands. Each command accepts *n* as a prefix, allowing you to move by *n* words rather than one. For each command the end of the line does not stop the movement of the cursor; instead, the cursor wraps around and continues counting words.

[w] or [W] Moves the cursor forward to the first character in the next word, where word is as defined above.

[e] or [E] Moves the cursor forward in the line to the last character in the next word, where word is as defined above.

[b] or [B] Moves the cursor backward in the line to the first character of the previous word, where word is as defined above.



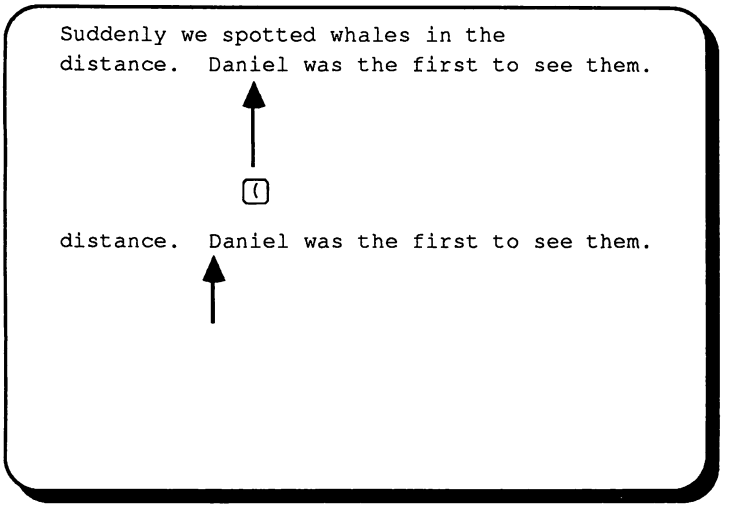
Moving the Cursor Sentence by Sentence

The *vi* editor also recognizes sentences. In *vi* a sentence ends in ! or . or ?. If these delimiters appear in the middle of a line, they must be followed by two blanks for *vi* to recognize them.

You can move the cursor from sentence to sentence in the file with the ((open parenthesis) and) (close parenthesis) commands. Each accepts *n* as a prefix, allowing you to move by *n* sentences.

(Moves the cursor to the beginning of the current sentence.

) Moves the cursor to the beginning of the next sentence.



Moving the Cursor Paragraph by Paragraph

Paragraphs are recognized by *vi* if they begin after a blank line. Each of the following paragraph movement commands accepts *n* as a prefix, allowing you to move by *n* paragraphs.

These are dummy spaces for new text.

{ Moves the cursor to the beginning of the current paragraph.

} Moves the cursor to the beginning of the next paragraph.

Moving the Cursor Within the Window

The *vi* editor has three commands for moving the cursor around the window. (See Figure 4-2).)

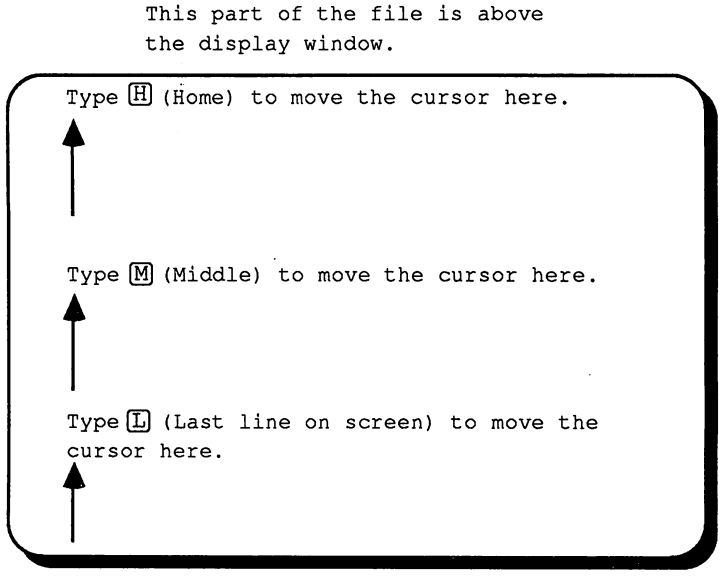
H Moves the cursor to the first line in the window.

M Moves the cursor to the middle line in the window.

L Moves the cursor to the last line in the window.

Moving the Cursor Outside the Window

vi allows you to move throughout your file, scrolling forward or backward, going to a specified line, or searching for a pattern.



Scrolling the Text

The following commands allow you to scroll the text of a file.

CTRL f — Scrolls forward one full screen.

CTRL d — Scrolls forward one half screen.

CTRL b — Scrolls backward one full screen.

CTRL u — Scrolls backward one half screen.

The CTRL f command scrolls the text forward one full window below the current window. To do this *vi* clears the screen and redraws the window. The two lines at the bottom of the current window are placed at the top of the new window. If there are not enough lines left in the file to fill the window, the screen displays a ~ (tilde).

The CTRL d command scrolls down a half screen to reveal text below the window. When you type CTRL d, the text appears to be rolled up at the top and unrolled at the bottom. The lines below the screen to appear on the screen, while the lines at the top of the screen disappear. If there are not enough lines in the file, a bell sounds.

The CTRL b command scrolls the screen back a full window to reveal the text above the current window. To do this, *vi* clears the screen and redraws the new window. Unlike the CTRL f command, CTRL b does not leave any reference lines from the previous window. If there are not enough lines above the current window to fill a full new window, a bell sounds and the current window remains on the screen.

The CTRL u command scrolls up a half screen of text to reveal the lines just above the window. When the cursor reaches the top of the file, a bell sounds to signify that you cannot move further.

Moving to a Specified Line

The G command positions the cursor on a specified line in the window; if that line is not currently on the screen, the G command clears the screen and redraws the window around it. If you do not specify a line, G moves the cursor to the last line of the file.

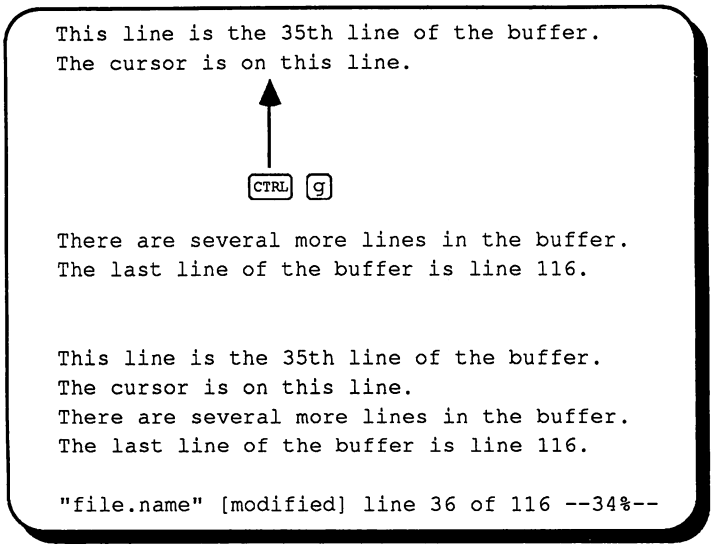
n G Moves to the *n* th line of the file.

G Moves to the last line of the file.

Line Numbers

Each line of the file has a line number corresponding to its position in the buffer. To get the number of a particular line, position the cursor on it and type CTRL g. A status notice appears at the bottom of the screen, telling you

- The name of the file.
- If the file has been modified.
- The line number on which the cursor rests.
- The total number of lines in the buffer.
- The percentage of the total lines in the buffer represented by the current line.



Searching for a Pattern of Characters: the / and ? Commands

The fastest way to reach a specific place in your text is by using the search commands: /, ?, n, or N. These commands allow you to search forward or backward in the buffer for the next occurrence of a specified character pattern.

/pattern Searches forward in the buffer for the next occurrence of the characters in *pattern*, and puts the cursor on the first of those characters. For example, the command line

/Hello world ↵

finds the next occurrence in the buffer of the words **Hello world** and puts the cursor under the **H**.

?pattern Searches backward in the buffer for the first occurrence of the characters in *pattern*, and puts the cursor on the first of those characters. For example, the command line

?data set design ↵

finds the last occurrence (before your current position) of the words **data set design** and puts the cursor under the **d** in **data**.

You can repeat the search commands by typing / or ? in succession. The editor remembers the last search request (using / or ?) and repeats it. For example, the command

/find ↵

moves the cursor down to the first occurrence of the word **find**. Then the command

/ ↵

moves the cursor down to the next occurrence of the word **find**.

/ and ? are both last line commands. They appear on the last line of your screen as you type them.

Alternatively, use the commands n and N to repeat the pattern search commands.

n	Repeats the last search command.
N	Repeats the last search command in the opposite direction.

The commands n and N do not appear on your screen as you type them.

The pattern search commands do not wrap around the end of a line to search for a pattern. For example, say you are searching for the words **Hello World**. If **Hello** is at the end of one line and **World** is at the beginning of the next, the search command does not find that occurrence of **Hello World**. They do, however, wrap around the end or the beginning of the buffer to continue a search. If you are near the end of the buffer, and the pattern for which you are searching (with the /*pattern* command) is at the top of the buffer, the command finds the pattern.

Note that the / and ? search commands do not allow you to specify particular occurrences of a *pattern* with numbers. You cannot, for example, request the third occurrence (after your current position) of a *pattern*.

Creating Text

The following commands allow you to enter text.

[a] Appends text after the cursor.

[A] Appends text at the end of the current line.

[i] Inserts text before the cursor.

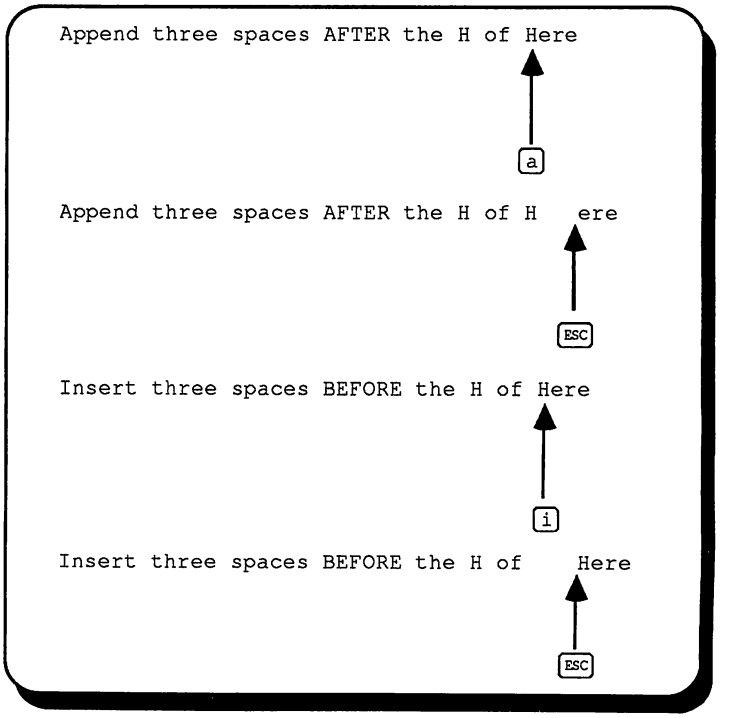
[I] Inserts text before the first non-blank character in the line.

[o] Creates a blank line below the current line and places you in input mode. You can issue this command from any position within the current line.

[O] Creates a blank line above the current line and places you in input mode. You can issue this command from any position within the current line.

When you have finished creating text with any of these commands, you press [ESC] to return to command mode.

The following examples compare the append and insert commands. The arrows show the position of the cursor.



You can delete text while in input or command mode, and you can undo entirely the effects of your most recent command.

Deleting Text in Input Mode

⌫ (backspace)	Deletes a single character.
@	Deletes all text entered on the current line since you last entered input mode. The characters are not actually erased until you press ESC.

Undoing the Last Command

u	Undoes the last command issued.
U	Nullifies all changes made to the current line as long as the cursor has not been moved from the line.

If you type u twice in a row, the second command undoes the first; your undo is undone! Say you delete a line by mistake and restore it by typing u. Typing u a second time deletes the line again.

Deleting Text in Command Mode

The following commands delete characters, words, paragraphs, and lines. Each command accepts *n* as a prefix, allowing you to change *n* elements of the text at once.

x	Deletes the character under the cursor.
dw	Deletes from the character under the cursor to the end of the current word including the spaces (if any) at the end of the word, or deletes one punctuation mark and the spaces (if any) that follow it.
d{	Deletes from the beginning of the current paragraph to the cursor.
d}	Deletes the character under the cursor to the end of the current paragraph.

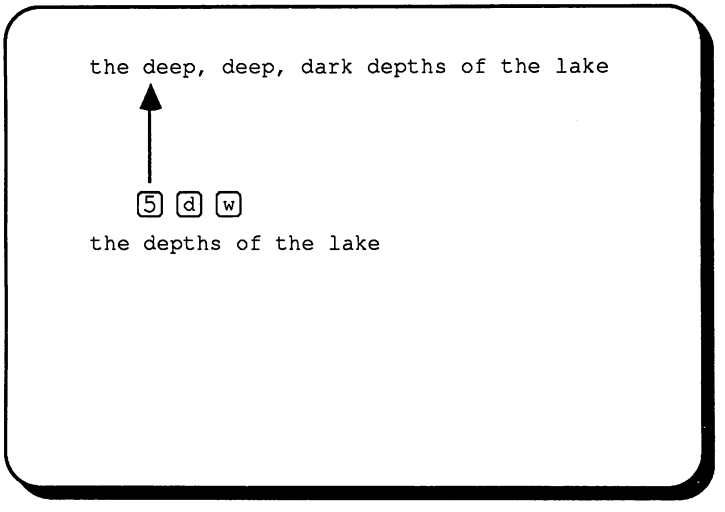
[d][d] Deletes the line with the cursor. If the prefix *n* is included *n* lines are deleted, beginning with the line containing the cursor. If *n* is greater than 5, *vi* displays this message on the bottom of the screen:

n lines deleted.

If you try to delete more lines than exist in the file, a bell sounds and no lines are deleted.

[D] Deletes from the character under the cursor to the end of the line.

To delete three words and two commas, type [5] [d] [w].



Modifying Text

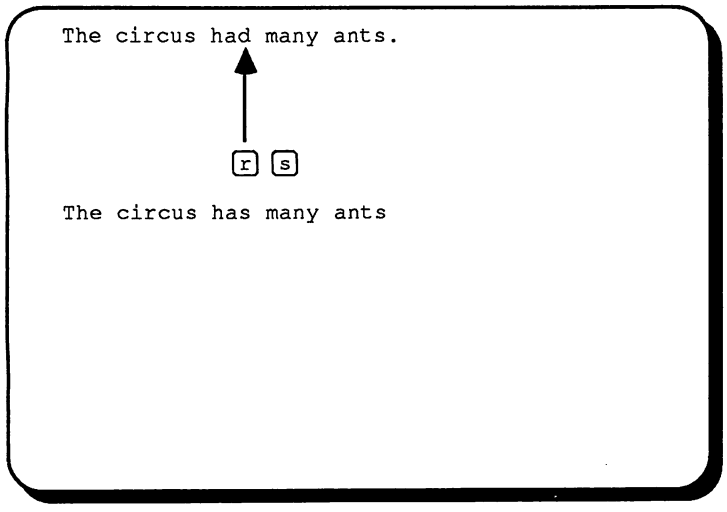
vi has commands that allow you to delete and create text simultaneously.

Replacing Text

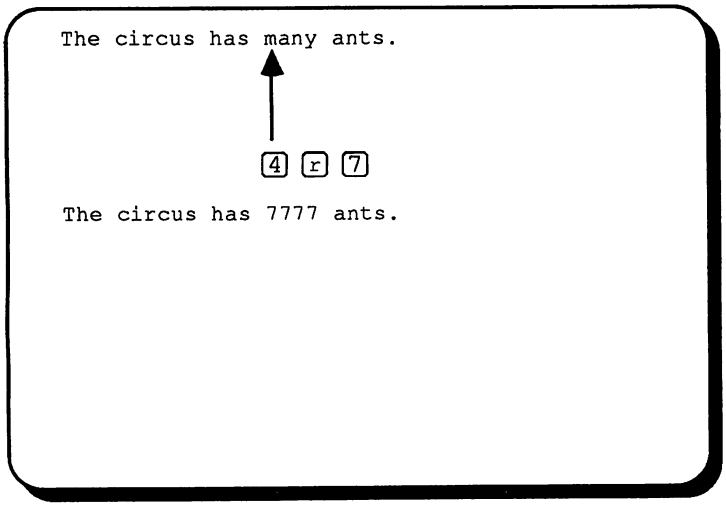
The replace commands replace one or more characters with new text.

[r] Replaces the current character (the character shown by the cursor). The [r] command does not initiate text input mode, so it does not need to be followed by [ESC]. To replace *n* characters, use the command with an *n* prefix. Again, no [ESC] is needed.

R Replaces characters typed over until ESC is pressed. If the end of the line is reached this command appends the input as new text.



To change many to 7777, type



Substituting Text

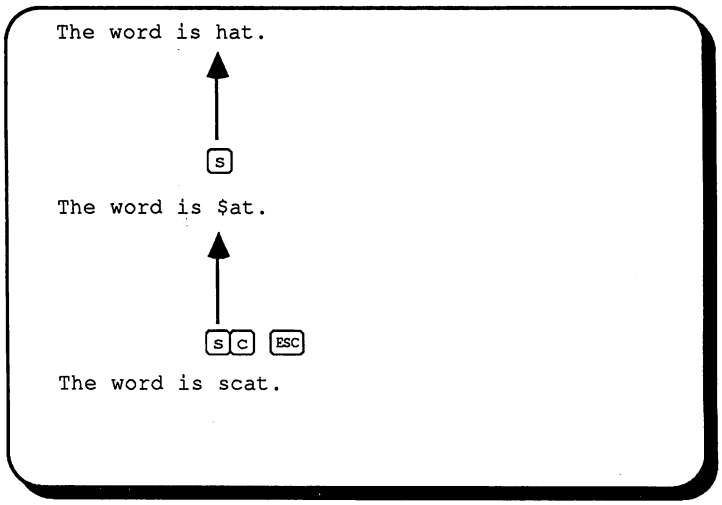
The substitute commands delete one or more characters and place you in text input mode. After you add text, press ESC.

s Deletes the character under the cursor and places you in text input mode.

[S] Deletes the entire line and places you in text input mode.

Each version of the substitute command accepts *n* as a prefix.

When you enter the [s] command, the last character in the string to be replaced is overwritten by a $ sign. Characters are not erased from the screen until you type over them or leave input mode (by pressing [ESC]).



Changing Text

The change commands delete one or more words, lines, sentences, or paragraphs and place you in input mode. After you change the text press [ESC] to leave input mode.

[c][w] Deletes a word or the remaining characters in a word and places you in text input mode.

[c][c] Deletes all the characters in the line and places you in text input mode.

[C] Deletes the character under the cursor to the end of the line and places you in text input mode.

[c][(] Deletes the characters from the left of the cursor to the beginning of the sentence and places you in text input mode.

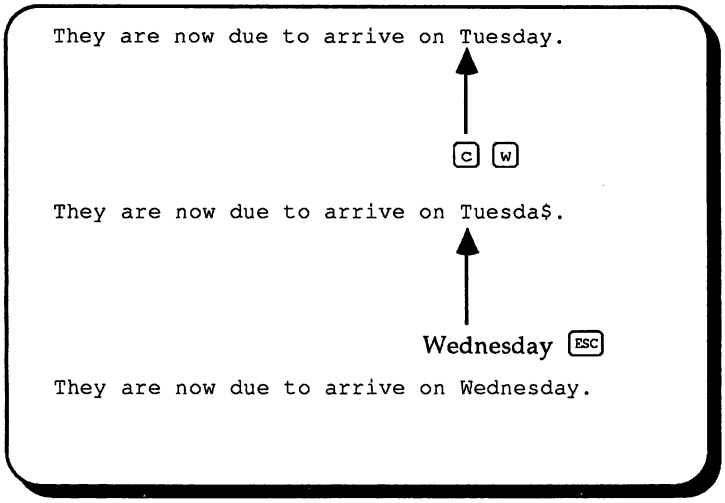
[c][)] Deletes the character under the cursor to the end of the sentence and places you in text input mode.

c{ Deletes the characters from the left of the cursor to the beginning of the paragraph and places you in text input mode.

c} Deletes the character under the cursor to the end of the paragraph and places you in text input mode.

Each of these commands accepts *n* as a prefix.

The c w and C commands use a $ sign to mark the last letter to be replaced.



Notice that the new word (`Wednesday`) has more letters than the word it replaced (`Tuesday`). Once you have executed the change command you are in text input mode and can enter as much text as you want. To return to command mode, press ESC.

The C command works in the same way.

Cutting And Pasting Text

vi provides a set of commands that cut and paste text within a file, or allow you to copy a portion of text and place it in another section of a file.

Moving Text

Whenever you delete text from the *vi* buffer the text is saved in a special temporary buffer until overridden by another deletion. This provides a way to move text from one place to another. The

[p] and [P] commands allow you to do this. The general procedure is to delete the text, move the cursor to the new location, and then issue the [p] or [P] command.

[p] Places the contents of the temporary buffer after the cursor. If one or more lines were deleted, this command places them below the line on which the cursor appears.

[P] Places the contents of the temporary buffer after the cursor. If one or more lines were deleted, this command places them above the line on which the cursor appears.

To position a partial sentence deleted with the [D] command into the middle of another line, position the cursor in the space between two words, then press [p] ([P]). The partial sentence is placed after (before) the cursor.

Remember to use the [p] and [P] commands right after a deletion, since the temporary buffer only stores the results of one command at a time.

Fixing Transposed Letters

A quick way to fix transposed letters is to combine the [x] and [p] commands as [x] [p]. [x] deletes the letter. [p] places it after the next character.

Notice the error in the next line.

```
A line of tetx
```

This error can be changed quickly by placing the cursor under the `t` in `tx` and then pressing the [x] and [p] keys, in that order. The result is

```
A line of text
```

Copying Text

The "yank" commands allow you to copy text to a temporary buffer and then place the contents of the buffer anywhere in the file. When you use the yank commands the original text remains undisturbed, and unlimited copies can be placed where you wish.

[y][w] Copies the character under the cursor to the end of the word into the buffer.

[y][y] Copies a line of text into the buffer.

Each command accepts an *n* prefix.

Once you have yanked text, use the [p] and [P] commands to put the text in the new position. Type [p] ([P]); the text appears on the line below (above). The temporary buffer retains the text you have placed there until you yank other text or quit *vi*.

Copying or Moving Text Using Registers

vi provides special registers to use when you need to move or copy several groups of text to different parts of the file. Using registers is useful if pieces of text must appear in many places in the file.

To store text in a register, you can either yank or delete the text. The stored text stays in the specified register until you leave *vi* or place new text in the same register.

To place text in a register use the following format.

"*xCOMMAND*

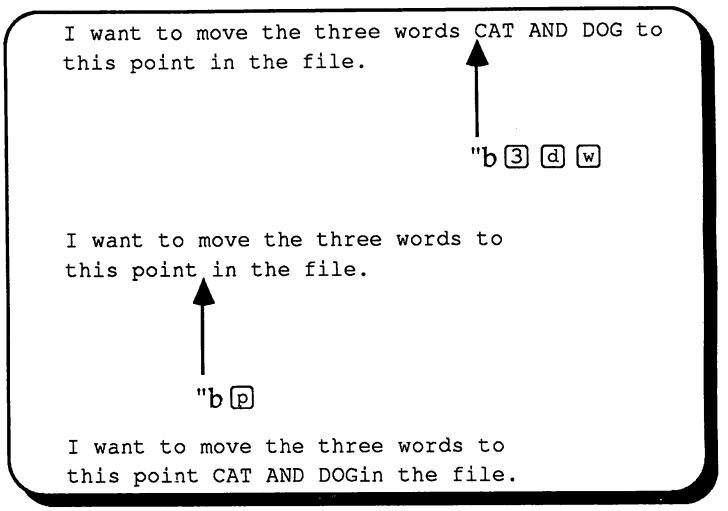
The double quote " signals that it is a register command. Register commands operate in command mode; they do not appear on the screen when you type them. The *x* is the name of the register and can be any single letter of your choice. *COMMAND* can be any of a variety of *vi* commands. For example, to yank two lines of a file and place them in register **a**, put the cursor at the beginning of the first line you want to yank and type

"a2[y][y]

The two lines of text remain in register **a** until you either exit from the editor or store new text in that register. To put a copy of the text in register **a** into the file, move the cursor to the desired location and type

"a[p]

In the next example, three words are deleted and placed in register **b**. They are then moved to another place in the file.



Other Commands

The following miscellaneous commands are useful.

.	Repeats the last command.
J	Joins two lines together.
CTRL l	Clears the screen and redraws it.
~	Changes lower case to upper case and vice versa.

Repeating the Last Command

Period (.) repeats the last command to create, delete, or change text in the file. It is often used with the pattern search commands.

For example, suppose you forgot to capitalize the S in United States. Use the command

/United states ↵

to find the first occurrence of the phrase in your file. Issue the replace command r to change **s** to**S**. Then type

/ ↵

to continue your search. When you find another occurrence simply type a period; *vi* remembers your last command and repeats the substitution of s for S.

Joining Two Lines

The [J] command joins the current line with the next line. To enter this command, place the cursor on the current line, and press [SHIFT] and [j] simultaneously.

For example, suppose you have the following two lines of text:

```
Dear Mr.
Smith:
```

To join these two lines into one, place the cursor under any character in the first line and type

[J]

You immediately see the following on your screen:

```
Dear Mr. Smith:
```

Notice that *vi* automatically places a space between the last word on the first line and the first word on the second line.

Clearing and Redrawing the Window

If another user sends you a message using the write command while you are editing with *vi*, the message appears in your current window and prints over part of the text you are editing. To restore your text after you have read the message,

(1) Press [ESC] to return to command mode.

(2) Then type [CTRL] [l]. *vi* erases the message and redraws the window exactly as it appeared before the message arrived.

Changing Cases

A quick way to change any lower case letter to upper case, or vice versa, is by putting the cursor on the letter to be changed and typing a [~] (tilde). You can change several letters by typing ~ several times, but you cannot precede the command with a number to change several letters with one command.

Using Line Editing Commands in *vi*

The *vi* editor has access to many of the commands provided by the **ed** line editor. (For a complete list of **ed** commands see the **ed**(1) page in the *Commands Reference Manual.*) This section discusses some of the **ed** commands that are particularly useful when using *vi.* They all begin with a with a : (colon) and operate in last line mode. In most cases the full command appears on the last line of your screen.

Temporarily Returning to the Shell: the :sh *and* :! *Commands*

When you enter *vi,* the contents of the buffer fill your screen, and you cannot issue any shell commands. If you wish to issue shell commands without quitting *vi,* you can use the **:sh** or **:!** commands. You can also use the **:!!** command to repeat the previous **:!** command.

:!*shell_command*
: Allows you to escape *vi* to run a single shell command. While in command mode type **:!**, followed immediately by the shell command. The shell runs your command, gives you output, and prints the message

[Hit return to continue]

When you press ↵ *vi* refreshes the screen and the cursor reappears exactly where you left it.

:!!
: Allows you to repeat the previous shell command. While in command mode type **:!!** and press ↵. The shell runs the command and gives the output of the previous **:!***shell_command.*

:sh
: Allows you to issue multiple commands before returning to *vi.* While in command mode type **:sh** and press ↵. A shell command prompt appears on the next line. Type your command(s) after the prompt as you would normally do while working in the shell. When you are ready to return to *vi,* type CTRL d or **exit**; your screen is refreshed with your buffer contents and the cursor appears where you left it.

Even if you change directories while temporarily in the shell, you can return to *vi* by typing CTRL d or **exit**.

Saving Changes or Writing Text to a New File: the :w *Command*

The **:w** (for write) command allows you to write text to a file by copying text from the *vi* buffer to the file. The general format of the command is

:line_number,line_number **w** *filename*

(The *line_number* arguments and *filename* are optional.)

The **:w** command has two important uses in *vi*:

- It allows you to save changes you have made to the *vi* buffer during an editing session.
- It allows you to create a file by copying lines of text from the file you are currently editing into a file that you specify.

The **:w** command insures that changes you have made to your file won't be lost in the event of a power interrupt or system failure. The command

:w ↵

writes the contents of the *vi* buffer to the file you are editing, saving any changes you have made since the beginning of the editing session or since the last **:w** command.

NOTE
To insure that significant changes won't be lost in the event of an interrupt, issue the **:w** command frequently during your editing session (for example, after every page).

You can also use the **:w** command to write a section of the buffer to a new file. For example, to write lines 23 through 37 of the buffer to a file named **newfile**, type

:23,37w newfile ↵

vi reports the successful creation of your new file with the following information:

```
"newfile" [New file] 10 lines, 265 characters
```

Finding the Line Number

To determine the line number of a line, move the cursor to it, then type

:= ↵

This command gives the line number at the bottom of the screen, then then returns the cursor to that point in the text.

You can move the cursor to any line in the buffer by typing : and the line number, as follows.

:*n* ↵

Deleting the Rest of the Buffer

One of the easiest ways to delete all the lines between the current line and the end of the buffer is by using the line editor command **d** with the special symbols for the current and last lines.

:.,$d ↵

. represents the current line; **$** the last line.

Adding a File to the Buffer

To add text from a file below a specific line in the editing buffer, use the **:r** (read) command. For example, to put the contents of a file called *data* into your current file, place the cursor on the line above the place where you want it to appear. Type

:r data ↵

You may also specify the line number instead of moving the cursor. For example, to insert the file *data* below line 56 of the buffer, type

:56r data ↵

Don't be afraid to experiment; u undoes **ed** commands as well as *vi* commands.

Global Substitution

Two special line editor commands, **s** (for substitute) and **g** (for global), together allow you change all occurrences of a character string with one command. For example, say you have several pages of text about the DNA molecule in which you refer to its structure as a helix. You want to change every occurrence of the word helix to double helix. The substitute and global commands allow you to do this with one command line.

The general form of the command line is

:*line_number,line_number***s/***old_text/new_text***/g** ↵

(The **line_number** arguments are optional; if they are omitted, the scope of the command is the current line.)

For example, to change helix to double helix throughout the file, type

:1,$s/helix/double helix/g ↵

Quitting vi

Use the following command sequences to quit the *vi* editor. Commands preceded by a colon (:) are line editor commands.

ZZ

:wq ↵

:x ↵

Writes the contents of the temporary buffer to the file currently being edited and quits *vi*.

:w *filename* ↵
Writes the temporary buffer to a new file named *filename*.

:w! *filename* ↵
Overwrites an existing file called *filename* with the contents of the buffer.

:q ↵
Quits *vi* without writing the buffer to a file. This works only if you have made no changes to the buffer; otherwise *vi* warns you that you must either save the buffer first with the **:w** command or use the **:q!** ↵ command to terminate.

:q! ↵
Quits *vi* without writing the buffer to a file and discards all changes made to the buffer.

Suppose you want to give your file a new name, *junk*. Type

:w junk ↵

After you write to the new file, leave *vi* by typing

:q ↵

If you try to write to an existing file, you receive a warning. For example, if you try to write to a file called *johnson,* the system responds with:

```
"johnson" File exists - use "w! johnson"
to overwrite
```

If you want to replace the contents of the existing file with the contents of the buffer, use the **:w!** command to overwrite *johnson.*

:w! johnson ↵

Your new file overwrites the existing one.

If you edit a file, make some changes to it, and then decide you don't want to keep the changes, or if you accidentally press a key that gives *vi* a command you cannot undo, leave *vi* without writing to the file. Type

:q! ↵

Special Options For vi

The *vi* command has some special options. It allows you to

- Recover a file lost by an interrupt to the operating system.
- Place several files in the editing buffer and edit each in sequence.
- View a file without making changes.

Recovering a File Lost by an Interrupt

If there is an interrupt or disconnect the system exits the *vi* command without writing the text in the buffer back to its file. The operating system, however, stores a copy of the buffer for you. When you log back in you can restore the file with the **–r** option for the *vi* command. Type

vi –r filename ↵

The changes you made to *filename* before the interrupt occurred are now in the *vi* buffer. You can continue editing the file, or you can write the file and quit *vi*.

Editing Multiple Files

If you want to edit more than one file in the same editing session, issue the *vi* command, specifying each file name. Type

vi file1 file2 ↵

vi responds by telling you how many files you are going to edit. For example:

```
2 files to edit
```

After you have edited the first file, use the **:w** command to write your changes to the file (*file1*).

Once this is done you can bring the next file into the editing buffer by typing

:n ↵

The system responds by printing a notice at the bottom of the screen telling you the name of the next file to be edited and the number of characters and lines in that file.

Viewing a File

It is often convenient to be able to inspect a file by using the *vi* editor's powerful search and scroll capabilities, while at the same time protecting yourself against accidentally changing the file during an editing session. To view a file in read-only mode invoke the editor as *view* rather than *vi*.

C. Resources

Chapter One of the *Programmer's Guide*, called *Creating & Maintaining Programs*, provides a complete primer to *vi* basics, *ex* commands, use of buffers, text block management, etc.

For a complete reference to *vi* commands see the *vi(1)* page in the *Commands Reference Manual*.

General Literature

Mohamed el Lozy: *Editing in a UNIX Environment*. Prentice-Hall, 1985.

D. Quick Reference

Command	Action
^-B*	move backward one page
^-D	scroll down
^-E*	expose one more line at bottom of screen
^-F*	move forward one page
^-G	print statistics on current file and position
^-H	move back one space
^-J	move to corresponding position on next line
^-L	redraw screen, useful if scrambled
^-M	same as CR
^-N	move to corresponding position on next line
^-P	move to corresponding position on previous line
^-R	redraw screen, eliminating deleted lines marked by @
^-U	scroll up
^-Y*	expose one more line at top of screen
^-Z*	suspend currently active UNIX command

* This command may not be available on some versions of UNIX.

Command	Action
A	append text at end of current line
B	move to start of current big word
CR	move to first non-blank position on the next line
C	change to end of line
D	delete to end of line
E	move to end of current big word
F*x*	move backward to character *x* on current line
*n*G	move to start of line *n*
H	move to start of top line on current screen
I	insert text at beginning of current line
J	join next line to current one
L	move to start of last line on current screen
M	move to start of middle line on current screen
N	search for pattern in opposite direction
O	insert line above current line
P	put contents of a buffer into text before cursor
R	replace characters by overtyping
S	change line
T*x*	move backward to character after *x* on current line
U	undo all changes on current line
W	move to start of next big word
X	delete character before cursor
Y	copy current line into buffer
ZZ	save file and quit

Command	Action
a	append text
b	move to beginning of word
c	change object
d	delete object
e	move to end of word
f*x*	move forward to character *x* on current line
h	move left one space
i	insert text
j	move down one line
k	move up one line
l	move right one space
m*x*	associate a mark *x* with current cursor position
n	search for pattern in same direction
o	insert below line
p	put contents of a buffer into text after cursor
r	replace characters
s	change characters
t*x*	move forward to character before *x* on current line
u	undo last change
w	move to start of new word
x	remove character
y	copy object into buffer
z	redraw screen around current line

Command	Action
!*object command*	send lines from current line to *object* to *command*, and replace them by its output
$	move to end of current line
%	if given with cursor on (,{, or [, moves it to matching), }, or]
"	return to start of line you were previously on
'*x*	move to start of line containing mark *x*
(	move to start of current sentence
)	move to start of next sentence
+	move cursor to first non-blank character on next line
-	move to first non-blank position on previous line
.	repeat last command which changed buffer
/*pat*CR	move forward to first occurence of pattern *pat*
0	move to start of current line
<	decrease indent of each level of object by one shiftwidth
=*	reindent lisp program, as if it had been entered
>	indent each line of object by one more shiftwidth
?*pat*CR	move backward to first occurence of pattern *pat*
[[	move to start of current section
n |	move to column *n* on current line
]]	move to start of next section
^	move to first non-blank character on current line
`*x*	go to position marked *x*
``	return to previous position
{	move to start of current paragraph
}	move to start of next paragraph
~*	change case of character if alphabetic

COMMUNICATIONS

CHAPTER FIVE

Contents

- *A. Session One*: Some basic communication commands.
- *B. Basics*: Brief explanation of remote login, copying files between various machines, executing commands remotely, sending mail and files.
- *C. Resources*: Where to find more information.
- *D. Quick Ref*: A list of the most frequently used communication commands.

Introduction

UNIX offers a choice of commands for communicating with other users and systems. You can exchange messages and files with other users (on either your own or another computer system), and execute commands on a remote system.

To help you take advantage of these capabilities, this chapter explains the following commands.

mail, uname, and **uuname**	For exchanging messages and files.
rlogin, rcp, and **rsh**	For use over an Ethernet or Cheapernet network: to log into a remote machine, copy files between machines, and execute commands on a remote machine.
ct and **cu**	For use with RS-232 connections (direct or via modem): to connect a remote terminal and to connect to a remote system.

A. Session One

To send mail

mail *addressee*

To read your mail

mail

To mail a file

mail *addressee* < *filename*

Remote login, remote copying

rlogin
rcp

B. Basics

To send and receive messages, use the **mail** command. It allows you to create and send messages, to manage incoming mail. The same program (**mail**) is also used to send files containing memos, reports, and so on.

Sending Mail to One Person

Here is the basic command line format for sending mail:

mail *option(s) login* ↵

where *login* is the recipient's login name. It can be either of the following: To send and receive messages you use the **mail** command. It allows you to create and send messages and manage incoming mail. You can also use **mail** to send files containing memos, reports, and so on.

Sending Mail to One Person

Here is the basic command line format for sending mail:

mail *option(s) login* ↵

where *login* is the recipient's login name. It can be either of the following:

- A login name if the recipient is on your system (for example, *bob*).

- A system name and login name if the recipient is on another system able to communicate with yours (for example, *sys2!bob*).

Though not required, various options are available with the **mail** command. One of these, **–s** *subject*, is described below.

Sending messages to yourself is a good way to experiment with the **mail** command. At the system prompt type

mail *your_login* ↵

The system now prompts you for a subject. Type in a subject of your choice, press ↵, and start typing the text of your message on the next line. There is no limit to the length of your message. When you have finished typing, send the message by typing ~. (tilde period) or CTRL d at the beginning of a new line.

The following example shows how this procedure appears on your screen.

```
Stardent 1500/3000: mail phyllis↵
Subject: Change of Schedule↵
My meeting with Smith's↵
group tomorrow has been moved↵
up to 3:00 so I won't be able to↵
see you then. Could we meet↵
in the morning instead?↵
EOT
Stardent 1500/3000:
```

Notice that the ~. or CTRL d do not appear on the screen. Instead **EOT** (end of text) prints. This is the system's indication that you have ended typing the message. The prompt on the last line means that your message has been queued (placed in a waiting line of messages) and will be sent.

(Various commands are available to allow you to edit messages you are writing, read in files as part of a message, and so on. See the **mail**(1) command description in the *Commands Reference Manual.*)

The mail is sent to a file with the same name as your login ID in the */usr/mail* directory, and you receive a notice that you have mail.

Sending mail to yourself can also serve as a handy reminder system. Suppose you (login ID *bob*) want to call someone the next morning. Send yourself a reminder in a mail message.

```
Stardent 1500/3000: mail bob↵
Subject: Reminder↵
Call Accounting and find out↵
why they haven't returned my 1985 figures!↵
EOT
Stardent 1500/3000:
```

When you log in the next day, a notice appears on your screen informing you that you have mail waiting to be read.

The **–s** *subject* option lets you specify the subject of the message on the command line. With this option there is no subject prompt, so you type your message immediately after issuing the **mail** command. The following example shows how this works.

```
Stardent 1500/3000: mail –s 'Picnic Friday' bob↵
Just to remind you, Bob, that↵
the picnic is on Friday. Bring your softball glove!↵
EOT
Stardent 1500/3000:
```

Note that the subject, *Picnic Friday*, has been enclosed in quotes. This is necessary if the subject has multiple words. Otherwise the system will try to interpret part of the subject as the recipient's login.

Undeliverable Mail

If you make an error typing the recipient's login the **mail** command cannot deliver your mail. Instead it prints a message telling you it has failed and that it is returning your mail. It returns your mail to a file called *dead.letter* in your home directory.

For example, say you (owner of the login *kol*) want to send a message to a user on your system with the login *chris*.

Failing to notice that you have incorrectly typed the login as **cris**, you try to send your message.

```
Stardent 1500/3000: mail cris↵
Subject: Meeting Change↵
The meeting has been changed to 2:00.↵
EOT
Can't send to cris
"/usr/kol/dead.letter" 1/38
Stardent 1500/3000:
```

The two lines following **EOT** give the system's response. The system confirms that the message couldn't be sent, that it has been saved in the file */usr/kol/dead.letter*, and that the text of the saved message has 1 line and 38 characters. You can now read what you intended to send, or mail the message again using file redirection (see *Sending Files* below in this chapter).

Sending Mail to Several People Simultaneously

You can send a message to several people at once by including all their login names on the **mail** command line. For example:

```
Stardent 1500/3000: mail tommy jane wombat dave↵
Subject: Diamond cutters!↵
The game is on for tonight at diamond three.↵
Don't forget your gloves!↵
Your Manager↵
EOT
Stardent 1500/3000:
```

Sending Mail to Remote Systems: the **uname** *and* **uuname** *Commands*

Until now we have assumed that you are sending messages to users on your own computer system. You are likely, however, to want to send mail to users on other systems both in your own building and elsewhere.

You send mail to users on other systems by adding the name of the recipient's system before the login ID on the command line. For instance:

mail sys2!bob ↵

Notice that the system name and the recipient's login ID are separated by an exclamation point.

Before you can run this command, however, you need three pieces of information:

- The name of the remote system.
- Whether or not your system and the remote system can communicate.
- The recipient's login name.

The **uname** and **uuname** commands allow you to find this information.

If you can, get the name of the remote system and the recipient's login name from the recipient. If the recipient does not know the system name, have her or him issue the following command on the remote system:

uname -n ↵

The command responds with the name of the system. For example:

```
Stardent 1500/3000: uname -n↵
squirrel
Stardent 1500/3000:
```

Once you know the remote system name, the **uuname** command can help you verify that your system can communicate with the remote system. At the prompt, type

uuname ↵

This generates a list of remote systems with which your system can communicate. If the recipient's system is on that list, you can send messages to it by **mail**.

You can simplify this step by using the **grep** command to search through the **uuname** output. At the prompt, type

uuname | grep *system* ↵

(Here *system* is the recipient's system name.) If **grep** finds the specified system name, it prints it on the screen. For example

```
Stardent 1500/3000: uuname | grep squirrel↵
squirrel
Stardent 1500/3000:
```

This means that *squirrel* can communicate with your system. If *squirrel* does not communicate with your system, **uuname** returns a prompt.

As an example, suppose you want to send a message to login *rocket* on the remote system *squirrel*. Verify that *squirrel* can communicate with your system and send your message. The following screen shows both steps.

```
Stardent 1500/3000: uuname | grep squirrel↵
squirrel
Stardent 1500/3000: mail squirrel!rocket↵
Subject: Writing seminar follow-up↵
Rocket,↵
The final counts for the writing seminar↵
Our department – 18↵
Your department – 20↵
Tom↵
EOT
Stardent 1500/3000:
```

Managing Incoming Mail

If you are logged in when someone sends you mail, the following message is printed on your screen:

> you have mail

This means that one or more messages are being held for you in a file called */usr/mail/your_login*, usually referred to as your mailbox. To examine these messages, type the **mail** command without any arguments:

> **mail** ↵

A message summary such as the following (for login *jane*) is printed on your screen.

```
Stardent 1500/3000: mail↵
mail version xx.xx 10/28/87  Type ? for help.
"/usr/mail/jane":  3 messages 2 new 1 unread
 U  1 phyllis          Tue Oct 14 1987 13:53    2/25   Weekly report due
>N  2 fred             Thu Oct 22 1987 09:10   2/109   Meet for lunch?
 N  3 tommy            Thu Oct 22 1987 11:01    2/94   Fred's visit
?
```

The summary shows that 3 messages have been saved in */usr/mail/jane*. Two of these, Messages 2 and 3, are marked with an N to show that they are new, while Message 1 is marked with a U to show that it is old but has not yet been read. The flag, or less than sign (>), beside message 2 shows that it is next in line to be read. Additional information includes the login of the sender (*phyllis, fred* or *tommy*), the date and time the message was received, the number of lines and characters (for example 2/25 in message 1 means 2 lines, 25 characters), and the subject.

The question mark (?) on the last line shows that you are in command mode and that the system is waiting for you to issue a command. Here is a list of useful commands; they are described below. In each case you have a choice of typing the whole word (for example **print**) or typing just the abbreviation shown in bold type (**p**).

print or **t**ype	prints (or types) the next message.
n or **p**rint *n* or **t**ype *n*	prints (or types) message number *n*.
next	prints the next message (used to step through the message list).
delete *n*	deletes message *n*. If the *n* is omitted the current message is deleted.a
save *n filename*	saves message **n** in the file *filename*. If *n* is not specified, the current or flagged message is saved. If *filename* is not specified, the message is saved in the file *mbox* in your home directory.
header	prints the message summary or header.
help	prints a summary of commands.
x or **e**xit	lets you leave mail without changing what is in */usr/mail/your_login*. Any deletes or saves are ignored.
quit	leaves mail, saving unread messages in */usr/mail/your_login* and read messages in *mbox* (in your home directory).

For instance, to print the next mail message you can type **p** or **t**, as in the following example.

```
?p↵
Message 2
From fred Thu Oct 22 09:10 PST 1987
To: jane
Subject: Meet for lunch?
Status: R

How about meeting for lunch when I come Tuesday for the seminar?
We can finalize changes on our joint paper.
?
```

The first two lines give the message number, the login of the sender (*fred*), and the date and time the message was sent. The next three lines give the login of the recipient (*jane*), the subject, and the status (??????). The text of the message follows, and the terminating question mark (?) shows that the system is ready for another command. Message 2 was printed because it was the first unread message in the list. (See *jane's* message summary above.)

If a long message is being displayed on your terminal screen, you may not be able to read it all at once. You can interrupt the printing by typing CTRL s. This freezes the screen, giving you a chance to read. When you are ready to continue, type CTRL q to resume printing.

You may also print messages by number. For example, the command **t** 3 will print or "type" message number 3. The "next" command **n** will print the next message, incrementing each time it is invoked. For example, if user *jane* types **n** after printing message 2, message 3 will appear.

```
?n↵
Message 3
From tommy Thu Oct 22 11:01 PST 1987
To: jane
Subject: Fred's visit
Status: R

While Fred is here, let's get together and discuss
the recent article in the AMS journal.  OK?

?
```

To delete the current, or flagged message, type **d** after the question mark. A question mark prompt will appear to show that the deletion has been completed and to await the next command. To delete message number *n* type **d** *n*. For example:

?d 3 ↵

deletes message number 3.

You can save messages in two ways. The command

?s *n filname* ↵

saves message number *n* in file *filename*. If *n* is omitted, the current or flagged message is saved. If *filename* is omitted, the file is saved in *mbox* (in your home directory). Here is an example.

```
?s3 tommy_note↵
"tommy_note" [new file] 9/174
?
```

The system verifies the creation of a new file *tommy_note* (in your current working directory) with 9 lines and 174 characters. (This count includes header information.) If the file *tommy_note* already existed, the message would have been appended to the end of that file. If you wish you can save a file in another directory by specifying a relative or full pathname. (For instructions see *Pathnames*, Chapter 3.)

For a full list of the **mail** options, type **help** in response to the **mail** ? prompt.

You can exit mail in two ways. The commands

?x ↵

or

?exit ↵

let you exit the command program, preserving all messages. Deleted and saved files remain in */usr/mail/your_login* to be viewed the next time mail is invoked. The command

?q ↵

lets you exit or "quit" mail, preserving only the unread messages in */usr/mail/your_login*. Messages that have been read are saved in the file *mbox* in your home directory. Here is an example.

```
?q↵
Saved one message in /usr/jane/mbox
Held two messages in /usr/mail/jane
Stardent 1500/3000:
```

Sending and Receiving Files Via the **mail** *Command*

The **mail** command is used for sending files either to your local system or to a remote system. To send a file in a **mail** message, you must redirect the input to that file on the command line. Use the < (less than) redirection symbol as follows:

mail *login* < *filename* ↵

(For further information see *Redirecting Input* in Chapter 5.) Here *login* is the recipient's login ID and *filename* is the name of the file you want to send. For example, if you (login *jane*) want to send a copy of a file called agenda to the owner of login *sarah* (on your system) type the following command line:

```
Stardent 1500/3000: mail sarah < agenda↵
Stardent 1500/3000:
```

The prompt that appears on the second line means the contents of agenda have been sent. When *sarah* issues the **mail** command to read her messages, she receives notice of the message.

```
Stardent 1500/3000: mail↵
mail version xx.xx 10/28/87  Type ? for help.
"/usr/mail/sarah":  1 message new
>N  1 jane              Thu Oct 22 1987 09:15   2/89
```

No subject was specified, so the subject field is left blank.

To send the same file to more than one user on your system, use the same command line format with one difference; in place of one login ID, type several, separated by spaces. For example:

```
Stardent 1500/3000: mail sarah tommy dingo wombat < agenda↵
Stardent 1500/3000:
```

Again, the prompt returned by the system in response to your command is a signal that your message has been sent.

The same general command line format can also be used to send a file to a user on a remote system that can communicate with yours. Simply specify the name of the remote system before the user's login name. Separate the system name and the login name with an ! (exclamation point) as follows:

mail *system!login* < *filename* ↵

For example:

mail squirrel!rocket < agenda ↵

If a file has been sent to you via the **mail** command, you receive notification that a message has been received. If you then request that the message be printed, the entire file appears on the screen. It can then be saved using the **save** command in **mail**. Of course, if you know a particular file is expected, you can issue the **save** command without first viewing the file. This is especially useful if a large file (many pages) is being sent. If you are sending a large file you may want to warn the recipient (in a separate **mail** message) that a large file is due.

Networking

Stardent 1500/3000 supports networking over an Ethernet or Cheapernet local area network or through an RS-232 port.

The following commands are used for Ethernet and Cheapernet communications:

rlogin Lets you log into a remote system.

rcp Lets you copy files from one system to another.

rsh Lets you execute a command on a remote system.

The following commands are used for RS-232 port communications:

ct Allows you to connect your computer to a remote terminal that is equipped with a modem.

cu Enables you to connect your computer to a remote computer.

These commands are described in the following sections. Before they can be used, however, your computer must be configured for Ethernet/Cheapernet or RS-232 port communications. See the *System Administrator's Guide* for information.

Ethernet/Cheapernet Communications: the rlogin Command

The **rlogin**(1c) command connects your terminal (on your local system) to a remote system. The command follows this format:

rlogin *remotehost options*

remotehost is the name of the remote system, as listed in the administration file */etc/hosts*. (See the *System Administrator's Guide* for more on the */etc/hosts* file.) The most common option is **-l** *username*, which lets you log into the remote system under a different user name. For instance, the command

rlogin meadow

logs you into the remote system *meadow* under your current login name, while the command

rlogin meadow -l moose

logs you into the remote system *meadow* under the login name *moose*. If you use the **-l** *username* option, the remote system prompts you for a password as follows:

```
Stardent 1500/3000: rlogin meadow -l moose
password: enter moose's password
meadow:
```

Ethernet/Cheapernet Communications: the rcp Command

The **rcp** command lets you copy files from one system to another. The format of the command is

rcp file1 file2

where each *file* is either a pathname on the local system (relative or absolute), or it is a remote filename of the form *remotehost:filename*.

Here are some examples. For each, assume that you are logged in as *squirrel*, on your local system, *forest*, and that you are in your home directory.

rcp memos/draft meadow:/usr/rjs/memos
: copies the file *draft* in the directory *memos* (in your home directory) to a file of the same name in the directory */usr/rjs/memos* on the system *meadow*.

rcp meadow:/usr/junk/example1 example1
copies the file *example1* in the directory */usr/junk* on the system *meadow* to a file of the same name in your home directory on your local system.

rcp meadow:~bjm/pingpong /games/pingpong
in C-shell, copies the file *pingpong* in the home directory of the user *bjm* on the system *meadow* to a file of the same name in the directory */games* on your local system. (In C-shell, the symbol ~ means "home directory of". If it is not followed by a user name, the symbol means "your home directory." The symbol is not recognized by the Bourne shell.)

rcp meadow:/work/draft1 field:/work/draft2
copies the file *draft1* in the directory *work* on the system *meadow* to a file named *draft2* in the directory *work* on the system *field*.

Ethernet/Cheapernet Communications: the **rsh** *Command*

The **rsh**(1c) command lets you execute a command on a remote machine. Here is the format:

rsh *remotehost command*

Here is an example:

rsh meadow ls -a /work
gives a listing of all the files in the directory */work* on the system *meadow*.

RS-232 Port Communications: the **ct** *Command*

The **ct** (connect) command connects your computer to a remote terminal equipped with a modem and allows a user on that terminal to log in. To do so, the command dials the phone number of the modem. The 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 it to log in on the computer.

To execute the **ct** command, follow this format:

ct *option(s) telno* ↵

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

Suppose you are logged in and you want to connect a remote terminal to your computer. The phone number of the modem on the remote terminal is 932-3497. Here is an example command line.

ct –h –w5 –s1200 9=9323497 ↵

ct calls the modem using a dialer operating at a speed of 1200 baud. If a dialer is not available, the **–w5** option causes **ct** to wait for a dialer for five minutes before quitting. The **–h** option tells **ct** not to disconnect the local terminal (the terminal on which the command was issued) from the computer. See **ct**(1) in the *Commands Reference Manual* for more on the **ct** command.

RS-232 Port Communications: the **cu** *Command*

The **cu** command connects a remote computer to your computer and allows you to be logged in on both computers simultaneously. This means that you can move back and forth between the two computers, transferring files and executing commands on both, without dropping the connection.

The method used by the **cu** command depends on the information you specify on the command line. You may specify the telephone number of the remote computer, in which case the number is passed to an automatic dial modem. Alternatively, your system may be set up to accept a system name on the command line. If so, **cu** obtains the phone number from a special *Systems* file. If there is a direct link to the remote computer (not involving a modem), the line or port associated with the direct link can be specified on the command line.

Once the connection is made, the remote computer prompts you to log in on it (you may need to press ↵ first). When you have finished working on the remote terminal, you log off and terminate the connection by typing ~. (tilde period). You are still logged in on the local computer.

To execute the **cu** command, follow this format:

cu *option(s)* *telno* ↵

or

cu *option(s)* *systemname* ↵

where the arguments are as follows:

telno
The telephone number of a remote computer.

Equal signs (=) represent secondary dial tones and dashes (–) represent four-second delays.

systemname
A system name that is listed in the *Systems* file. To see the list of computers in the *Systems* file, you can run the **uuname** command. (See the *System Administrator's Guide* for more on the Systems file.)

The **cu** command obtains the telephone number and baud rate from the *Systems* file and searches for a dialer.

Once your terminal is connected and you are logged in on the remote computer, all standard input (input from the keyboard) is sent to the remote computer. Here are the commands you can execute while connected to a remote computer through **cu**.

~.	Terminate the link.
~!	Escape to the local computer without dropping the link. To return to the remote computer, type CTRL d.
~!*command*	Execute *command* on the local computer.
~$*command*	Run *command* locally and send its output to the remote system.
~%**cd** *path*	Change the directory on the local computer where *path* is the pathname or directory

name.

~%**take** *from* [*to*] — Copy a file named *from* (on the remote computer) to a file named *to* (on the local computer). If *to* is omitted, the *from* argument is used in both places.

~%**put** *from* [*to*] — Copy a file named *from* (on the local computer) to a file named *to* (on the remote computer). If *to* is omitted, the *from* argument is used in both places.

~%**break** — Transmit a break to the remote computer (can also be specified as ~%**b**).

NOTE
The use of ~%**take** requires the existence of the **echo** and **cat** commands on the remote computer. Also, **stty tabs** mode should be set on the remote computer if tabs are to be copied without expansion.

The use of ~%**put** requires **stty** and **cat** on the remote computer. It also requires that the current erase characters on the remote computer be identical to the current ones on the local computer.

Suppose you want to connect your computer to a remote computer called *eagle*. The phone number for eagle is 847–7867. Enter the following command line:

cu –s1200 9=8477867 ↵

The **–s1200** option causes **cu** to use a 1200 baud dialer to call *eagle*. If the **–s** option is not specified, **cu** uses a dialer at the default speed for the modem being used.

When *eagle* answers the call, **cu** notifies you that the connection has been made, and prompts you for a login ID:

```
connected
login:
```

Enter your login ID and password. (Depending on the remote computer, you may need to press ↵ before you can be prompted to login.)

The **take** command allows you to copy files from the remote computer to the local computer. Suppose you want to make a copy of a file named *proposal* for your local computer. The following command copies *proposal* from your current directory on the remote computer and places it in your current directory on the local computer. If you do not specify a file name for the new file, it is also called *proposal.*

~%take proposal ↵

CAUTION
Be careful about your files when using ~%**take** and ~%**put**. Both commands will overwrite existing files of the same name.

The **put** command allows you to do the opposite: copy files from the local computer to the remote computer. Say you want to copy a file named *minutes* from your current directory on the local

computer to the remote computer. Type

~%put minutes minutes.9–18 ↵

In this case, you specified a different name for the new file (*minutes.9-18*). Therefore the copy of the *minutes* file that is made on the remote computer is called *minutes.9-18*.

C. Resources

The on-screen *man* help facility has information about various communication commands. Use *man mail* for mail options.

General Literature

Frey & Adams: *A Directory of Electronic Mail Addressing and Networks*.O'Reilly & Associates, 1989.

Kochan & Wood: *UNIX Networking*. Hayden Books, 1989.

Ricken & Weiman: *Introduction to UNIX Networking*. .sh consulting inc. 1989.

Andrew S. Tanenbaum: *Computer Networks/* (Second Edition). Prentice Hall, 1988.

D. Quick Reference

Operation	Command
Access mail messages	**mail**
Connect local system to remote systems	**rlogin** *remotehost options*
Connect to both local and remote systems	**cu** *option(s) telno*, or **cu** *option(s) systemname*
Copy files from one system to another	**rcp** *file1 file2*
Determine system name	**uname -n**
Execute command on remote system	**rsh** *remotehost command*
Remote connection with modem	**ct** *options(s) telno*
Send mail to yourself	**mail** *your_login*
Send mail to others	**mail** *option(s) login*
Send mail to several people simultaneously	**mail** *login1 login2*
Send mail to remote system	**mail** *system!login_name*
Send file through mail	**mail** *login < filename*
Send file through mail to several people	**mail** *login1 login2 < filename*
Send file to remote system	**mail** *system!login < filename*
Verify local and remote system communication	**uuname -n**

GETTING STARTED IN PROGRAMMING

CHAPTER SIX

Contents

- A. Session One: Refer to Chapter 7.
- B. Basics: About Stardent 1500/3000's programming tools.
- C. Resources: Where to find more information.
- D. Quick Ref: A list of essential commands and concepts.

Introduction

This chapter introduces Stardent 1500/3000's programming tools. They range from basic programming tasks, such as compiling programs, to sophisticated techniques for optimizing code.

The topics presented in this chapter include:

- *fc* and *cc* - the and C compilers
- Command line options
- Compilation control statements
- Compiler directives
- *dbg* - the debugger
- *nm, od, prof, mkprof,* and *size* - debugging tools
- *ld* - the linker
- Archive libraries
- Code optimization

A. Session One

To start using the programming tools presented in this chapter, you should see them used in context, so no *A. Session One* section is provided for this chapter. Instead, refer to Chapter 7, *Example Sessions*.

B. Basics

To create programs for Stardent 1500/3000, you need to know how to compile programs and use the compilation tools that are specific to the Stardent 1500/3000 and each programming language. It is also helpful to learn the basics of the available debugging tools. Information for using the Linker and archive libraries is provided for those who wish to search and/or create non-standard libraries. The last topic in this section, *Code Optimization*, provides the basics for making your code efficient and fast.

Compiling

The syntax for the compiler command is as follows.

fc [*options*] [*filespec*] [*options*] [*filespec...*]

NOTE
Options and file names may be intermingled.

where

options
is a set of options.

filespec
is a UNIX file path by which a source file may be accessed. The compiler can handle a mix of different programming languages and object files on the same command line. For example, file names can end with the characters *.f*, *.c*, *.o*, *.a*, or *.s*. If the compiler does not know how to create or to access a file you have specified, it generates an error message.

The syntax for the C compiler command is as follows.

cc [*options*] [*filespec*] [*options*] [*filespec...*]

NOTE
Options and file names may be intermingled.

where

options
is a set of options.

filespec
is a UNIX file path by which a source file may be accessed.

Either command has the facilities to compile the program and link it with default libraries. A run file is produced from the compilation and link, always called *a.out*, unless you have overridden the default file name. You may now run your program by typing

```
prompt> a.out ↵
```

on the command line. The results of your program are displayed on the terminal.

Command Line Options

Command line options are used on the command line of a compilation command. All options are case-sensitive, and some options have defaults and negative forms. Following are some of the most often used options for *cc* or *fc*. Refer to the *Programmer's Guide* for additional discussion and a complete list of options.

Preprocessor Options

-E
Output expanded source to standard output. This option stops the compilation process after all macros and conditional compilation expressions have been expanded.

-I*dir*
Search for include files in directory *dir*.

Compiler Options

-full_report
Invoke the vector reporting facility. This option produces a report, in Fortran-like notation, of how code is vectorized, what code the compiler generated and why.

-g
Generate information for the Stardent 1500/3000 debugger.

-O3
Perform common subexpression elimination, instruction

scheduling, vectorization and parallelization.

-subcheck

Produce code to check at runtime to ensure that each array element accessed is actually part of the appropriate array.

Loader Options

-l*tag*

Search a library called *libtag.a*. The default is to search first in */lib* and then in */usr/lib*.

-o

Change the output filename from *a.out* to a specified name.

Compilation Control Statements

In addition to compilation control options specified on the command line, you can communicate with the Stardent 1500/3000 preprocessor by including certain statements in the source code, such as these:

#define

Used to assign values to symbols used in the source code or to act as preprocessor symbols to control the inclusion of other source code.

#elif and *else*

Allow you to include alternative source code if tested conditions are found to be false.

#endif

Terminates an *#if*, *#ifdef* or *#ifndef* statement allowing formation of nested conditions.

#if

Lets you apply integer arithmetic tests against the integer value of a symbol and include or not include sections of source code based on results of the test.

#ifdef and *ifndef*

Test whether or not you have defined a particular symbol.

#include

Defines the name of a file to be copied into the source code and to be processed with the file.

#line

To label lines of source text.

Compiler Directives

This section briefly discusses the role of compiler directives, but does not provide details of the directives themselves.

Stardent 1500/3000 provides restructuring compilers, which can often determine at compilation time the optimal run-time use of parallel and vector hardware.

The user may have certain information regarding the use of the code at run-time which may not be obvious in the code or simply not discernable to the compiler at the time of compilation. By using certain compiler directives, it is possible to provide these key bits of information that could allow (or force) the compiler to choose a more efficient algorythm, thus overriding the compiler's natural, more cautious approach.

Compiler directives can do the following, which, in turn, causes the compiler to generate more efficient code:

- Force vectorization or parallelization of a particular loop.
- Indicate the best loops to vectorize or parallelize.
- Request that a loop be executed exactly as written.
- Request that a procedure be in-lined within a loop.
- Declare a procedure that can take vectors as arguments.

The Stardent 1500/3000 Debugger

A debugger is a program revealer, used to figure out what happened to the code. When any behavior in your program is different from what you expect, use a debugger. There are generally four circumstances when you use a debugger.

(1) The program does not work and you need to determine the location of an error in the program logic.

(2) The program works but the answers are not what you expected. You must isolate the differences between your expectations and the results.

(3) The program works, but you want a post mortem on what happened during execution.

Do not confuse using a debugger with the need to include the *-g* option in the compilation of your program. The Stardent 1500/3000 compilation system usually provides more than adequate information to debug your program without the *-g* option. In fact, using this option may have undesirable effects unless used in the proper circumstances. Refer to the *Programmer's Guide* for more information on the *-g* option.

If you have determined that you want to use the debugger (*dbg*), here is a summary of its most significant abilities:

- The "master" instructions in the debugger reflect the position or operation of the code in the IPU.
- The debugger allows you to ask what the FPU is doing.
- The debugger can tell you when a process goes parallel and what the threads are doing.
- The debugger can be used to keep track of the synchronization between the IPU and the FPU.
- The debugger can be used to grab any process, and then look at it, stop it, kill it, or take control of it.

dbg can be used for the same tasks as any other debugger would be, such as controlling the execution of processes, setting break points, examining the contents of memory and registers and gaining source and object file information. For more information on the specifics of using *dbg*, refer to the *Programmer's Guide*.

dbg Tasks

To invoke *dbg*, the command is

dbg [*a.outfile*] [*corefile*]

To get online help in dbg, the command is

help [*keyword*]

To exit dbg, the command is

exit
or
finish
or
quit

Debugging Tools

There are several debugging tools available to you in addition to *dbg*. This section briefly discusses what these tools are and why you might want to use them.

nm Displays the symbol table from your *a.out*. For each symbol in the table the following information is displayed: storage class, type, size in bytes, source line number at which the symbol is defined, and the object file section containing the symbol. There are also many options for the command controlling the output. This may help you in organizing and sorting external and static symbols.

od Dumps and displays a file in one or more formats. These formats allow interpretation of certain data types in forms that include hexadecimal, octal, signed and unsigned decimal.

prof and *mkprof*
Produce a report on the amount of execution time spent in various portions of your program and the number of times each function is called. This is useful if your program is producing the desired results but is running very slowly.

mkprof
Requires no recompilation of the program. Produces profile reports for programs that have previously been compiled. Use *prof* when first compiling your program. Make sure to specify the **-p** option. See Chapter 9 of the *Programmers Guide* for details.

size Defines the amount of space that the three sections of a common object file (text, data and bss) will occupy (in bytes) when that file had been loaded, ready to run.

Linker

The basic function of the linker, also known as the link editor, is to combine object files into an executable program. The linking function is performed automatically when either the *cc* or *fc* command is invoked. You may, however, invoke the linker directly with the *ld* command.

The link editor performs the following functions: combines several object files into one, performs relocation, resolves external symbols, incorporates startup routines, and supports symbol table information.

There are quite a few options that can be used with the *ld* command. One of the most commonly used options is **- l***tag*. It directs the link editor to search a library named *libtag.a*; it is used to bring in libraries that are not normally in the search path.

For a more complete discussion of the available options, refer to the *Commands Reference Manual.*

Archive Libraries

A library is a collection of related object files (and possibly declarations). It is common in UNIX system computers to keep object files in archives. These archive files then hold object modules that make up a library. The (archive) library can be named on the *ld* command line, or with a link editor option on a compiler command line. This naming causes the link editor to search the symbol table of the archive file(s) when attempting to resolve references.

To create an archive file, use the *ar* command. Refer to the *Commands Reference Manual* for the details of all options.

Code Optimization

The Stardent 1500/3000 and C compilers automatically optimize programs, searching for speed increases. Programs which use vector instructions or can execute in parallel threads on several processor units can achieve substantial speed increases. This section provides a summary of the programming techniques you can use that take the best advantage of the automatic optimizations of the compilers.

- Use iterative loops wherever possible. The compiler operates more efficiently on iterative loops than on **DO WHILE** constructs.
- Convert branches into structured **IF-THEN-ELSE** statements instead of using **GO TO** statements.
- Make nested loops into larger single loops. Put as much calculation inside the loops as possible, to minimize synchronization.
- Do not unroll loops to enhance performance.
- Use language intrinsics, such as **MAX** and **MIN**, instead of writing code to test and then branch.
- Program for memory access and not to save memory.
- Do not program to avoid computing null elements in sparse arrays.
- Do not reference temporary variables outside of the loops in which they are used.
- Specify array dimensions as longest dimension first to shortest dimension last in , and do the opposite in C.
- Use **COMMON** and **EQUIVALENCE** statements only when absolutely necessary.
- Use compiler directives to handle loops with hidden bounds.

Refer to the *Programmer's Guide* for details of the previous summary.

C. Resources

Use the following list to find more information on specific topics.

- *fc* and *cc* - *Programmer's Guide*, Chapter 2, the *Commands Reference Manual*, and the on-line *man* pages
- Command line options - *Programmer's Guide*, Chapter 2
- Compilation control statements - *Programmer's Guide*, Chapters 2, 4, and 9

- Compiler directives - *Programmer's Guide*, Chapters 2, 6, and 11
- *dbg*, the debugger - *Programmer's Guide*, Chapters 4 and 5, and the on-line help facility within *dbg*
- *nm,od,prof,mkprof*, and *size* - *Programmer's Guide*, Chapter 4, the *Commands Reference Manual*, and the on-line *man* pages
- *ld*, the linker - *Programmer's Guide*, Chapter 3, the *Commands Reference Manual*, and the on-line *man* pages
- Archive libraries - *Programmer's Guide*, Chapter 3
- Code optimization - *Programmer's Guide*, Chapters 4, 6, 7, 9

D. Quick Reference

The Five Commandments of good programming:

- Use iterative loops wherever possible
- Use *do(while)* or *for(while)* statements
- Do not use *go to* statements
- Make nested loops into larger single loops
- Do not unroll loops

Compiler Optimization

Directive	Result
ASIS	loop not touched by vectorizer
INLINE	inline a named function
IPDEP	ignore dependencies that prevent parallelization
IVEP	ignore dependencies that prevent vectorization
NO_PARALLEL	don't run next loop in parallel
NO_VECTOR	don't run next loop in vector
PBEST	indicates the best loop to parallelize
VBEST	indicates the best loop to vectorize
PPROC	compile procedure for parallel execution
VPROC	function has assembly version taking vector args
VREPORT	invokes vector reporting on next loop
SCALAR	run next loop in scalar

Fortran Compiler Options

Invoke compiler with

fc [options][filespec][options][filespec]...

Option	Result
-43	compile and link for BSD execution
-all_doubles	set floating points to REAL*8
-blanks72	pad with blanks on right to column 72
-c	compile only, do not link
-catalog=*name*.in	create a database of inlined functions
-continuations=n	specify number of continuation lines
-cpp	invoke the C preprocessor
-cross-reference	generate cross-reference listing
-debug	add debug data to object file
-double_precision	use double precision for undeclared variables
-d_lines	compile debug lines starting with D or d
-extend_source	extend statement field to column 132

Option	Result
-fast	optimize but lose some precision
-full_report	detailed vectorizer report
-fullsubcheck	check each array subscript
-g	add debugging information
-I	suppress default search for included files
-I*dir*	search for included files in *dir*
-i	supress production of #ident information
-implicit	untype all variables
-include_listing	add included fields to listing file
-inline	inline functions
-i4	interpret INTEGER and LOGICAL as *4
-L	do not search -l libraries in /lib or /usr/lib
-messages	print warning messages
-Npaths=*name*.in	use inlined functions in *name*.in
-no_assoc	turn off association of variables
-no_directive	do not apply directives during compilation
-novector	do not generate vector code for loops
-O	synonymous with -O1
-O0	turn off optimizations
-O1	common subexpression elimination
-O2	do -O1 and vectorization
-O3	do -O2 and parallelization
-object	generate an object file
-onetrip	DO loops execute at least once
-opct	count of FPU ops executed dynamically
-P	preprocess only
-p	load with profiling information
-ploop	profile loops separately
-S	generate assembly source file
-s	strip line numbers and symbol table
-safe=procs	compile procedure for parallel execution
-safe_strings	correct for mismatched string parameters

Option	Result
-save	all variables declared are saved
-standard	check standard Fortran 77 usage
-subcheck	check each linear array subscript
-t	turn off certain warnings
-V	print version information
-verbose	generate message output
-vreport	report on attempt to vectorize
-vsummary	report on vectorized loops
-w	suppress warning messages

C Compiler Options

Invoke compiler with

cc [options][filespec][options][filespec]...

Option	Result
-43	compile and link for BSD execution
-c	compile only, do not link
-catalog=*name*.in	create a database of inlined functions
-full_report	detailed vectorizer report
-fullsubcheck	check each array subscript
-g	add debugging information
-Npaths=*name*.in	use inlined functions in *name*.in
-O	ignored
-O0	turn off all optimizations
-O1	common subexpression elimination
-O2	do -O1 and vectorization
-O3	do -O2 and parallelization
-opct	count of FPU ops executed dynamically
-w	suppress compiler warnings
-P	preprocess only

Option	Result
-p	load with profiling information
-ploop	profile loops separately
-S	generate assembly source file
-safe=loops	ignore dependencies for loop vectorization
-safe=parms	do not change parameter values
-safe=ptrs	do not change pointer values
-subcheck	check each linear array subscript
-T*hhhhhhh*	generate *a.out* with text address at *hhhhhhh*
-V	print version information
-v	use verbose message output
-vector_c	use -safe=loops and -safe=parms options
-vreport	report on attempt to vectorize
-vsummary	report on vectorized loops
-w	suppress warning messages at link and compile
-x	preserve global symbols only
-y*name*	print uses and definitions of *name*

C Preprocessor Options

Option	Result
-D*name*	*name*=1 to the preprocessor
-D*name=val*	*name=val* to the preprocessor
-E	output expanded source
-I	suppress default search for included files
-I*dir*	search for included files in *dir*
-i	supress production of #ident information
-U*name*	undefine *name*

Loader Table Options

Both *fc* and *ec* accept loader options.

Option	Result
-B*hhhhhhh*	generate *a.out* with bss address at *hhhhhhh*
-D*hhhhhhh*	generate *a.out* with data address at *hhhhhhh*
-e*sym*	default entry point has address *sym*
-L*dir*	search for -l libraries in *dir*
-list	generate listing file
-l*tag*	search library lib*tag*.a
-M	produce a load map
-m	generate simple load map
-N	generate NMAGIC file type
-n	suppress standard C startup routine
-o *filename*	output goes into *filename*
-r	produce a relocatable output file
-s	strip line numbers and symbol table
-T*hhhhhhh*	generate *a.out* with text address at *hhhhhhh*
-t	turn off certain warnings
-x	preserve global symbols only
-y*name*	print uses and definitions of *name*

EXAMPLE SESSIONS

CHAPTER SEVEN

This chapter provides an example with a cross-section of programming tools being used to create a single simple example. Following the simple example is a listing of a considerably more complex program that is modeled after the basic window program that you can find in Adrian Nye, *Xlib Programming Manual*, Vol. I. Sebastopol, CA: O'Reilly & Associates, Inc., 1989.

The difference between the example presented here and that in the X book is twofold:

- Here the X-Window is created by calling the function XCreateWindow instead of by using the function XCreateSimpleWindow. This allows the example to create a single-buffered full color X Window 24-planes deep (allowing a choice of over 16 million colors) on a system that might have only the basic graphics board (no graphics expansion board) installed. The default for X Windows on a system that does not have a graphics expansion board installed is to produce a pseudo color display (that is, only 8 planes deep), allowing a choice of only 256 different colors. This alternate function, XCreateWindow, makes it possible for this example to function on both a non-expanded and an expanded graphics system with no modifications.

- The original example draws text and some lines and waits for any key press to exit. This modified example draws a fixed number of random colored random positioned lines in a string-art layout in the window and exits on its own after the drawing has been completed and after a short sleep period has elapsed.

Note that it is entirely possible that the example program contains materials that are not needed for the function to be performed. For example, the function "TooSmall" is not used in this example. However this example is provided here since it is not obvious from the existing documentation on X how to create this special

effect (a full color, single buffered X Window on a system that only has a basic graphics board installed.).

A Sample Session

In creating a program, particularly a large one, it is highly unlikely that it will run correctly the first time through. And, even if it does run to completion, the program may not necessarily yield the correct results. It is often necessary to examine intermediate data values to determine that the appropriate logic is being followed and that the intermediate values make sense.

If a debugger program is not available, many programmers find that they have to install print statements at strategic points in their program. One might insert print statements for progress messages such as "got this far," or "it has not crashed yet," or "finished phase 1". Likewise, if certain data values will have an effect on the result, the programmer might check occasionally whether these values are within range: "Value of a is: XXX," and so on.

As the debugging effort progresses, it may occur to the programmer to check more and more data values or to run the program up to a certain point and deliberately cause an exit (something prior to a crash, for example). This often requires several passes through the program editor and compiler, and sometimes these efforts can be rewarded with the notification by the compiler of a typographical error in the debugging statements themselves, causing additional delay in the process.

However, with a debugger available, the programmer can enjoy greater productivity. Breakpoints can be established in many different places in the program. The program can be single-stepped after a breakpoint and the values of variables and intermediate results can be immediately examined without the need to edit and recompile, thereby increasing productivity and decreasing the time needed to fix the program.

The Stardent 1500/3000 system, running X Window, offers yet another advantage that improves productivity. This is the fact that two (or more) windows can be used simultaneously to work on the same problem in a multi-tasking (or multi-user) operation. For example, three windows can be opened, two of the windows fairly large, and one fairly small. All three windows are currently changed to the same directory, perhaps named "testdir." You could have one window running the program text editor, one

window ready to run the compiler, and a third window running the debugger. Thus by simply moving the mouse from one window to another, you could make and save a change to the program, compile the program, then debug the executable. The alternative, of course, is to do everyting in a single window, but if you can open more than one window, each with its own operating characteristics, why not take advantage of it?

Sample Application

In this basic program we use a subroutine and some data to print along with the some text that identifies the source of the data. The program is compiled so that the Stardent 1500/3000 debugger, *dbg*, can be used to examine the program variables and modify them before the data is actually printed.

This section consists of explicit instructions for:

- Creating a directory in which to store the sample program
- Creating three windows with which to run this example
- Moving into that directory with all three windows
- Starting *vi* in one window to create the program
- Typing in the program
- Compiling the program so that the debugger can be used to symbolically reference the program variables.
- Running the program without the debugger
- Starting the debugger
- Installing a breakpoint
- Running the program to the breakpoint
- Single stepping
- Viewing and changing the value of program variables
- Exiting the debugger

Creating The New Windows

This example assumes that you have the default environment, namely X Window, running on your Stardent 1500/3000. Here are the steps necessary to create and position three new windows onscreen.

1. Move the mouse until the onscreen pointer becomes an "X." That is, move it to a position in the background field of the display not currently occupied by a window.

2. Press and hold the rightmost mouse button until a menu appears. The top item on that menu should read "New Window".

3. With the mouse button held down, move the pointer until it causes that topmost item to be highlighted. That is, the topmost item becomes a different color than the other items on that menu.

4. Release the mouse button. This causes the New Window item to be selected.

5. In the upper lefthand corner of the display, you will soon see a notation such as "0x0" and the cursor character becomes a pair of lines, one pointing down and the other pointing to the right from a single intersection of the lines. Move the mouse until this new cursor is positioned where you would like a new window's upper lefthand corner to appear.

6. Press and release the leftmost button of the mouse to complete the creation of the new window. You have created a new window of the default size, normally 80x24 (80 characters across, by 24 lines long, where the physical size depends on the characteristics of the default font).

7. To move the window once it has been created, move the mouse cursor somethere within the window boundaries. Press and hold down the "LEFT" key on the keyboard, and then press and hold down the middle mouse button. Move the mouse cursor to the new position you wish the window to occupy. Finally release both the key and the mouse button.

8. To resize the window once it has been created, move the mouse cursor somewhere within the window boundaries. Press and hold down the "LEFT" key on the keyboard, and then press and hold down the right mouse button. Move the mouse cursor to make the window larger or smaller. The text that appears in the upper left corner of the display while you are performing this action tells you how many characters wide and tall the new window size will be.

(1) Repeat steps 1-8 to create and position a total of three windows. The first two windows should be at least 80x24. This chapter refers to them as Windows 1 and 2. The third window (called Window 3) will be used to run the compiler. It should be at least 80x4.

Creating The Sample Directory

Move the mouse cursor into each of these newly created windows, and into each, type the command:

```
cd
```

This moves each window into the user's home directory. When a new window is created, this is normally the default, however this simply assures that consistent results can be obtained.

Now, in any one of the three windows, issue the command:

```
mkdir testdir
```

This creates a test directory in which program examples can be placed and from which they can be run.

Move the mouse cursor to activate each window in turn and type the command for each window:

```
cd testdir
```

Now all three windows are operating in the same directory allowing shared access to that directory from all three locations.

Editing a Program

Move the mouse cursor to Window 1 (one of the 80x24 windows).

1. Type the following command:

   ```
   vi test1.c
   ```

   This opens the *vi* editor.

2. Change *vi* from command mode into insert mode by pressing

```
i
```

to be in insert mode.

3. Type the following program, exactly as presented here. If you make a mistake typing, use the backspace key to backspace and correct your error. If you need to know more about how to use *vi*, refer to Chapter 1 of the Programmer's Guide. That chapter contains a thorough tutorial for *vi*. As you complete each line, press RETURN to enter it and go on to the next line of the program.

```
/* test.c*/
main()
{
    int k;
    float x;
    double y;
    k = 10;  x = 1.25;  y = 3.4d-9;
    doprint(k, x, y);
}
doprint(m, n, p)
int m;
float n;
double p;
{
    printf("integer value is: %d0, k);
    printf("single precision value is: %f0, x);
    printf("double precision value is: %g0, k);
    return(0);
}
```

4. Now press the ESC key to end the insert mode of *vi* and exit.

5. Save the text by entering

```
ZZ
```

This ends the data entry phase of this project. Leave the *vi* editor window open in case you wish to do any more editing later.

Compiling The Program

This section compiles the program with the **-g** option so that it may be used with the debugger. Move the mouse cursor to the smallest of the three windows (Window 3) to run the compiler. Enter the command:

```
cc -g -o test test.c
```

This creates an executable file named *test* in the current directory. If the program generates any errors during the compile phase, examine and correct the errors in the editor window, save an edited copy by using the command "w! test.c" from command mode, and recompile until no errors occur. Again, refer to Chapter 1 of the *Programmer's Guide* for information about editing with *vi* if it is needed.

Running The Program

Move to Window 2 (the 80x24 window that is not running *vi*). Type the command:

```
test
```

You should obtain the following results:

```
integer value is: 10
single precision value is: 1.25
double precision value is:  0.0000000034
```

Running Under Control Of The Debugger

As mentioned above, you might find it necessary to check (and possibly change!!) intermediate results. Here is the method you can use to run the program under the control of the debugger *dbg* to obtain that capability, without recompiling your program.

1. Enter the following command to this same window (Window 2):

   ```
   dbg test
   ```

The debugger will start.

2. Enter this command to *dbg*:

   ```
   break in doprint
   ```

   This establishes a breakpoint just ahead of the first executable statement in the doprint subroutine.

3. Lets also establish a break in the main function:

```
break in main
```

4. Start the program running to the first breakpoint. Enter the command:

```
run
```

dbg tells you where it stopped.

5. While you are debugging a program, you can see where you are working in your source code by asking *dbg* to list your program source code:

```
list
```

This shows the source code listing with a set of double-arrows indicating the source code statement that will be executed next.

6. You can examine the contents of variables by simply typing their names:

```
k
x
y
```

Notice that at this first breakpoint, (which occurs before any of the executable statements in this function) no value has been assigned to any of those three variables. Thus you see the value that has, as a default, been assigned to all of these variables by the compiler (the value is zero).

7. Cause *dbg* to execute a single line of the source code by issuing the command:

```
step
```

8. Examine the contents of the variables. Type their names:

```
k
x
y
```

Notice that when the `step` command was executed, *dbg* printed the source code line that it executed. Notice that the variables now have the value that your program source code has assigned them.

9. Determine the status of the program to make sure that the breakpoints are where you expected them to be. Issue the command:

```
status
```

This shows you that there are two breakpoints established, one in main, and the other in doprint.

10. Continue to run the program by issuing the command:

```
cont
```

dbg continues the program and stops at the next breakpoint.

11. At this point you can do a traceback of the program to find out where it is now and where it came from if there are nested subroutine calls in effect. You could go back (see chapter 5 of the *Programmer's Guide* for information about *dbg*) to prior subroutines and examine the value of variables within those subroutines if there was a question in your mind as to how a calculation or a logical step got to this point. (You can only debug subroutines that are currently active... inactive subroutines have no context and therefore their variables have no value). You do a traceback by issuing the command `traceback` or the command `where`. Both are equivalent. The display shows you which subroutines have been called and the values of the parameters with which they have been called.

12. The program is now in the doprint function. You can now view the values of the local variables m, n, and p by typing their names. They are the same values as those in the main program.

```
m
n
p
```

13. You can modify the values of the variables by setting them equal to a different value. Note that the data type of the value you choose must match the data type in the program. Try these examples, modifying the data value, then type its name to confirm that the value has been accepted.

```
m = 15
n = -2.4e5
p = 1.9d-9
m
n
p
```

14. Continue the program to completion by entering the command:

```
cont
```

Notice that the printf statement this time outputs the values to which the variables have been set instead of those values passed into the program by the main function. This confirms that *dbg* was indeed capable of changing the way your program worked without editing and compiling again. This shows that you can use the debugger to increase your productivity.

15. Exit the debugger by entering the command:

```
exit
```

Or you can use the command `quit` if you wish.

A Sample X Program

Here is the source code for a sample program in X. The program opens a 24-plane X window on the display, draws a number of random colored random string-art lines, waits for a few seconds, then exits. Most of the drawing functions are contained in the subroutine *place_graphics*.

This program is based on the "baswin.c" program from the (*Xlib Programming Manual.*

The differences are in the way the X Window is opened (as indicated in the beginning of the chapter) and in the use of a different function to place the graphics (draws multicolored string art). The example is provided to show how to get a 24-plane screen on a system that has only a base graphics board.

To compile this program, enter the following command line:

```
cc -o stringart -lX baswin24.c
```

To run the program, just type:

```
stringart
```

The output is a set of multicolored interconnected lines. The program exits automatically.

```
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Xos.h>

#include <stdio.h>

/*      "@(#)icon_bitmap        1.2 Stellar 88/12/14"
                        Copyright 1988 by
                        Stellar Computer Inc.
                        All Rights Reserved

        This software comprises unpublished confidential information of
        Stellar Computer Inc. and may not be used, copied or made
        available to anyone, except in accordance with the license
        under which it is furnished.
*/
#define icon_bitmap_width 40
#define icon_bitmap_height 40
static unsigned char icon_bitmap_bits[] = {
   0xc3, 0xc3, 0x7f, 0x00, 0x78, 0x00, 0x00, 0x00, 0x00, 0xc0, 0x00, 0x00,
   0x00, 0x00, 0x80, 0x38, 0x00, 0x40, 0x00, 0x80, 0x24, 0x00, 0x00, 0x00,
   0x80, 0x44, 0x00, 0x00, 0x00, 0x80, 0x44, 0x00, 0x00, 0x00, 0x80, 0x74,
   0x00, 0x0f, 0x0c, 0x00, 0x7c, 0x3e, 0x41, 0x0e, 0x00, 0x44, 0x22, 0x41,
   0x02, 0x00, 0x84, 0x22, 0x46, 0x02, 0x00, 0x9c, 0x26, 0xcc, 0x02, 0x00,
   0x78, 0x3c, 0xcd, 0x36, 0x80, 0x00, 0x20, 0x06, 0x0c, 0x80, 0x01, 0x00,
   0x00, 0x00, 0x80, 0x01, 0x02, 0x40, 0x00, 0x80, 0x01, 0x06, 0x40, 0x00,
   0x80, 0x01, 0x04, 0x20, 0x00, 0x80, 0x01, 0x04, 0x20, 0x01, 0x80, 0x01,
   0x04, 0x20, 0x00, 0x80, 0x01, 0x04, 0x22, 0x00, 0x80, 0x01, 0x04, 0x33,
   0xf1, 0x81, 0x01, 0x88, 0x12, 0x31, 0x03, 0x01, 0x88, 0x12, 0x11, 0x02,
   0x00, 0x88, 0x12, 0x11, 0x02, 0x00, 0x48, 0x1a, 0x11, 0x02, 0x00, 0x70,
   0x04, 0x19, 0x82, 0x01, 0x00, 0x00, 0x00, 0x80, 0x01, 0x00, 0x00, 0x38,
   0x80, 0x01, 0x00, 0x00, 0xce, 0x80, 0x01, 0x00, 0x00, 0x83, 0x81, 0x81,
   0x07, 0x80, 0x01, 0x81, 0xe1, 0x04, 0xc0, 0x00, 0x83, 0x31, 0x08, 0x40,
   0x00, 0x82, 0x10, 0x08, 0x20, 0x00, 0x82, 0x19, 0x10, 0x30, 0x00, 0x86,
   0x0c, 0x30, 0x18, 0x00, 0x84, 0x04, 0x60, 0x0e, 0x00, 0xdc, 0x02, 0x80,
   0x03, 0x00, 0x70, 0x00, 0x00, 0x00, 0x00, 0x00};

#define BITMAPDEPTH 1
#define SMALL 1
#define OK 0
#define LINEWIDTH 2

/* These are used as arguments to nearly every Xlib routine, so it saves
 * routine arguments to declare them global.  If there were
 * additional source files, they would be declared extern there. */
Display *display;
int screen;

unsigned short xsubi[3] = { 0x123, 0x4567, 0x89ab };
```

```
void main(argc, argv)
int argc;
char **argv;
{
        Colormap cmap;
        XColor screen_def_return,
               exact_def_return;
        unsigned long black_pixel,
                      white_pixel;
        XVisualInfo *xvi;
        int nitems;
        XVisualInfo xvitemplate;
        Visual *visual;

        XSetWindowAttributes winattr;
        int valuemask = 0;

        Window win;
        unsigned int width, height, x = 0, y = 0;
        /* window size and position */
        unsigned int border_width = 4;  /* four pixels */
        unsigned int display_width, display_height;
        char *window_name = "stringart";
        char *icon_name = "";
        Pixmap icon_pixmap;
        XSizeHints size_hints;
        XEvent report;
        GC gc;
        XFontStruct *font_info;
        char *display_name = NULL;
        int window_size = 0;     /* OK, or too SMALL to display contents */
        Region region;           /* for exposure event processing */
        XRectangle rectangle;    /* for exposure event processing */

        printf("Program as written exits on its own\n");
        printf("after drawing a few screens of stringart\n");
        printf("Takes less than 30 seconds to run\n");
        printf("(Ignores all X events; draws lines, then exits\n");

        /* connect to X server */

        if ( (display=XOpenDisplay(display_name)) == NULL )
        {
                (void) fprintf( stderr,
                 "basicwin: cannot connect to X server %s\n",
                    XDisplayName(display_name));
                exit( -1 );
        }

        /* get screen size from X display structure function */
        screen = XDefaultScreen(display);
        display_width = DisplayWidth(display, screen);
        display_height = DisplayHeight(display, screen);

        /* note that x and y are 0, since the default position of the window
         * is the top left corner of the root. This is fine since the window
```

```
 * manager often allows the user to position the window before
 * mapping it. */

/* size window with enough room for text */
width = display_width/2, height = display_height/2;

xvitemplate.depth = 24;
xvitemplate.screen  = screen;
xvitemplate.class = DirectColor;
xvi = XGetVisualInfo(display, VisualScreenMask|VisualClassMask|
        VisualDepthMask, &xvitemplate, &nitems);
visual = xvi->visual;

if ( nitems == 0 ) {
        printf("no DirectColor visual\n");
        exit(1);
}

/* get default colormap */

cmap = XTitanDefaultDirectColormap(display,screen);

/* get black and white pixel values for chosen color map */

if (XAllocNamedColor(display, cmap, "black", &screen_def_return,
                &exact_def_return)) {
        black_pixel = screen_def_return.pixel;
} else {
        fprintf(stderr, "could not allocate black pixel\n");
        exit(1);
}

if (XAllocNamedColor(display, cmap, "white", &screen_def_return,
                &exact_def_return)) {
        white_pixel = screen_def_return.pixel;
} else {
        fprintf(stderr, "could not allocate white pixel\n");
        exit(1);
}


winattr.background_pixmap = None;
winattr.background_pixel = black_pixel;
winattr.border_pixel = black_pixel;
winattr.colormap = cmap;

valuemask = CWBackPixel | CWBorderPixel | CWColormap;


win = XCreateWindow(display, RootWindow(display,screen),
        x, y, width, height, border_width, 24, InputOutput,
        visual, valuemask, &winattr);


/* Get available icon sizes from Window manager
if (XGetIconSizes(display, RootWindow(display,screen),
```

```
   &size_list, &count) == 0)
{
        (void) fprintf( stderr,
        "Window mgr didn't set icon sizes - using default.\n");
        icon_width = 40;
        icon_height = 40;
}
else
{
        while (icon_width < size_list->min_width
        icon_width = size_list->max_width;
        icon_height = size_list->max_height;
}
all the icon size stuff commented out */

/* Create pixmap of depth 1 (bitmap) for icon */
icon_pixmap = XCreateBitmapFromData(display, win,
   icon_bitmap_bits, icon_bitmap_width, icon_bitmap_height);

/* Set resize hints */
size_hints.flags = PPosition | PSize | PMinSize;
size_hints.x = x;
size_hints.y = y;
size_hints.width = width;
size_hints.height = height;
size_hints.min_width = 600;
size_hints.min_height = 400;

/* set Properties for window manager (always before mapping) */
XSetStandardProperties(display, win, window_name, icon_name,
    icon_pixmap, argv, argc, &size_hints);

/* Select event types wanted */
XSelectInput(display, win, ExposureMask | KeyPressMask |
  ButtonPressMask | StructureNotifyMask);

/* create region for exposure event processing */
region = XCreateRegion();

load_font(&font_info);

/* create GC for text and drawing */
getGC(win, &gc, font_info);

/* Display window */
XMapWindow(display, win);

/* get events, use first to display text and graphics */
while (1)  {
        XNextEvent(display, &report);
        switch  (report.type) {
        case Expose:
                /* if this is the last contiguous expose
                 * in a group, draw the window */
                if (report.xexpose.count == 0) {
                        /* if window too small to use */
```

```
                        while (XCheckTypedEvent(display, Expose, &report));
                                if (window_size == SMALL)
                                        TooSmall(win, gc, font_info);
                                else {
                                        /* place graphics in window, */
                                        place_graphics(win, gc, width, height,
                                        font_info);
                                }
                        }
                        break;
                case ConfigureNotify:
                        /* window has been resized, change width and
                         * height to send to place_text and place_graphics
                         * in next Expose */
                        width = report.xconfigure.width;
                        height = report.xconfigure.height;
                        if ((width < size_hints.min_width) ||
                            (height < size_hints.min_height))
                                window_size = SMALL;
                        else
                                window_size = OK;
                        break;
                case ButtonPress:
                        /* trickle down into KeyPress (no break) */
                case KeyPress:
                        XUnloadFont(display, font_info->fid);
                        XFreeGC(display, gc);
                        XCloseDisplay(display);
                        exit(1);
                default:
                        /* all events selected by StructureNotifyMask
                         * except ConfigureNotify are thrown away here,
                         * since nothing is done with them */
                        break;
                } /* end switch */
        } /* end while */
}

getGC(win, gc, font_info)
Window win;
GC *gc;
XFontStruct *font_info;
{
        unsigned long valuemask = 0; /* ignore XGCvalues and use defaults */
        XGCValues values;
        unsigned int line_width = LINEWIDTH;
        int line_style = LineSolid;
        int cap_style = CapButt;
        int join_style = JoinMiter;
        int dash_offset = 0;
        static char dash_list[] = {12, 24};
        int list_length = 2;

        /* Create default Graphics Context */
        *gc = XCreateGC(display, win, valuemask, &values);
```

```
        /* specify font */
        XSetFont(display, *gc, font_info->fid);

        /* specify black foreground since default may be white on white */
        XSetForeground(display, *gc, BlackPixel(display,screen));

        /* set line attributes */
        XSetLineAttributes(display, *gc, line_width, line_style,
           cap_style, join_style);

        /* set dashes to be line_width in length */
        XSetDashes(display, *gc, dash_offset, dash_list, list_length);
}

load_font(font_info)
XFontStruct **font_info;
{
        char *fontname = "9x15";

        /* Access font */
        if ((*font_info = XLoadQueryFont(display,fontname)) == NULL)
        {
                (void) fprintf( stderr, "Basic: Cannot open 9x15 font\n");
                exit( -1 );
        }
}

#define RangeRand(xx) nrand48(xsubi)%(xx)

place_graphics(win, gc, window_width, window_height, font_info)
Window win;
GC gc;
unsigned int window_width, window_height;
XFontStruct *font_info;
{
        int x1, y1;
        int x2, y2;
        int width, height;
        int midx, midy;
        int i,j;
        int colorvalue, redvalue, greenvalue, bluevalue;

        height = -2 + window_height;
        width  = -2 + window_width;
        midx   = width/2;
        midy   = height/2;

        x1 = RangeRand(width);
        x2 = RangeRand(width);
        y1 = RangeRand(height);
        y2 = RangeRand(height);

        for(j=0; j<20; j++)
        {
            for(i=0; i<20; i++)
            {
```

```
                redvalue   = RangeRand(256);
                bluevalue  = RangeRand(256);
                greenvalue = RangeRand(256);
                colorvalue = 0x0 | (redvalue << 16)|
                    (greenvalue << 8)|bluevalue;

                XSetForeground(display, gc, colorvalue);
                XDrawLine(display, win, gc, x1, y1, x2, y2);
                XDrawLine(display, win, gc, width - x1, y1,
                            width - x2, y2);
                XDrawLine(display, win, gc, width - x1, height - y1,
                                            width - x2, height - y2);
                XDrawLine(display, win, gc, x1, height - y1, x2,
                            height - y2);
                x1 = x2;
                y1 = y2;
                x2 = RangeRand(width);  y2 = RangeRand(height);
            }
            sleep(1);
            XClearWindow(display,win);
        }
        /* these are not NORMALLY here,
            I just put it here to exit quickly */
        XUnloadFont(display, font_info->fid);
        XFreeGC(display, gc);
        XCloseDisplay(display);
        exit(1);
}


TooSmall(win, gc, font_info)
Window win;
GC gc;
XFontStruct *font_info;
{
        char *string1 = "Too Small";
        int y_offset, x_offset;

        y_offset = font_info->max_bounds.ascent + 2;
        x_offset = 2;

        /* output text, centered on each line */
        XDrawString(display, win, gc, x_offset,
           y_offset, string1, strlen(string1));
}
```

INVENTORY

APPENDIX A

System Documentation

Start Here: User's Guide (Part Number 340-0027-01)

Site Preparation Manual (Part Number 340-0023-03)

Installation/Administration Guide (Part Number 340-0119-00)

UNIX

Commands Reference Manual (Part Number 340-0103-00)

Programmer's Reference Manual (Part Number 340-0121-00)

Programmer's Reference Manual (Part Number 340-0122-00)

Documenter's Workbench (Part Number 340-0034-02)

General Literature

S.R. Bourne: *The UNIX System.* Addison-Wesley, 1982.

Kaare Christian: *The UNIX Operating System.* John Wiley & Sons, 1983.

Fiedler & Hunter: *UNIX System Administration.* Hayden Books, 1986.

Kerninghan & Pike: *The UNIX Programming Environment.* Prentice-Hall, 1984.

McGilton & Morgan: *Introducing the UNIX System.* McGraw-Hill, 1983.

Prata & Martin: *UNIX System V Bible.* Howard W. Sams & Co., Inc., 1987.

Mark G. Sobell: *A Practical Guide to the UNIX System.* Benjamin/Cummings, 1984.

Thomas & Yates: *A User Guide to the UNIX System.* OSBORNE/McGraw-Hill, 1982.

Waite, Martin, Prata: *UNIX Primer Plus.* Howard W. Sams & Co., Inc., 1983.

X Window System

Window System Manual (Part Number 340-0114-00)

Window System Toolkit (Part Number 340-0112-00)

Xlib Reference Manual (Part Number 340-0022-02)

General Literature

Oliver Jones: *Introduction to the X Window System.* Prentice-Hall, 1989.

Adrian Nye, *Xlib Programming Manual,* Vol. I. Sebastopol, CA: O'Reilly & Associates, Inc., 1989.

Scheifler, Gettys, Newman: *X Window System C Library and Protocol Reference.* Digital Press, 1988.

X Window System User's Guide. O'Reilly & Associates, Inc., 1988.

Communications

Network File System Manual (Part Number 340-0126-02)

General Literature

Frey & Adams: *A Directory of Electronic Mail Addressing and Networks.* O'Reilly & Associates, 1989.

Kochan & Wood: *UNIX Networking.* Hayden Books, 1989.

Ricken & Weiman: *Introduction to UNIX Networking.* .sh consulting inc. 1989.

Andrew S. Tanenbaum: *Computer Networks/* (Second Edition). Prentice Hall, 1988.

Programming

Programmer's Guide (Part Number 340-0116-00)

C: a Reference Manual (Part Number 340-0027-01)

Fortran Reference Manual (Part Number 340-0027-01)

General Literature

Steve Talbott: *Managing Projects with Make.* O'Reilly & Associates, Inc., 1988.

Graphics

Dore Programmer's Guide (Part Number 340-0107-00)

Dore Reference Manual (Part Number 340-0108-00)

INDEX

A

.awmrc 2:11
absolute pathname 3:5
accessing files 3:18
adding file to buffer 4:32
adding text 4:7–8, 10
alias feature 3:81, 84
archive files 6:8
archive libraries 6:1–2, 8, 10
argument definition 1:6
ASCII 1:2; 3:12, 38, 42
assign directory permissions 3:30
assign file permissions 3:30
AT&T 1:3

B

background window 2:4
Berkeley 4.3 UNIX 1:3
Bourne shell 1:5; 5:14
box, close 2:5
  junction 1:1, 3
  knob 1:3
  resize 2:5
buffer, adding file to 4:32
  deleting rest 4:32
byte 3:14–15

C

.cshrc 3:84, 89–90
C compiler options 6:13
C preprocessor options 6:14
C programming language 1:5
cases, changing 4:29
change directory permission 3:18, 30
change file permission 3:18, 30
changing cases 4:29
changing current directory 3:15
changing directories 3:2
changing permissions 3:33, 35–36
changing text 4:24
character, pattern matching one 3:42
characters, control 1:9, 14
  literal 1:9; 3:45
  pattern matching 3:41
  special 1:9; 3:43–10, 45, 49, 54
  transposing 4:26
Cheapernet communication 5:13
child directory 3:8, 12, 16
clearing window 4:29
close box 2:5
code optimization 6:1–2, 8, 10
codes, return 3:56, 66–67, 75
combine object files 6:8
command definition 1:6
command interpreter 1:5; 3:1, 39; 4:3
command line options 6:1, 3, 9
command line syntax 1:6–7
  repeating last 4:28
command substitution 3:51, 63, 65, 79
  undoing last 4:21
commands, cursor movement 4:6–7
  line editor 3:83; 4:30–84, 32–33
  mail message 5:8
  online 1:6
  redefining 3:81, 84
communication, Cheapernet 5:13
  Ethernet 5:13
compilation control statements 6:1, 4, 9
compiler directives 6:1, 5, 9–10
compiler optimization 6:11
conditional loops 3:72

conditions, testing loop 3:74–75
configuration, terminal 1:1; 4:5
constructing multiple conditional formats 3:75
contents, listing directory 3:2, 11
  page through 3:18, 20
control characters 1:9, 14
copy file 3:18, 25, 92
  hard 3:18
  printer 3:18
copying files system to system 5:12–13
copying text 4:26, 31
count characters in file 3:18, 29
count lines in file 3:18, 29
count words in file 3:18, 29
creating a directory to execute programs 3:57
creating a file with vi 4:4
creating a simple shell program 3:56
creating directories 3:10
creating text 4:5
C-shell 1:5; 3:1, 14, 26–27, 40, 81–84, 89; 4:3; 5:14
C-shell programming language 3:82, 89
current directory 3:4, 7–9, 11–13, 15, 27, 29–30, 36, 41, 53, 56–57, 74, 79; 5:17; 7:7
current working directory 3:4–7, 15; 5:10–16
cursor 1:9; 2:4; 4:5–27, 29–32, 37; 7:5–39, 7
cursor movement commands 4:6–7
  moving 4:10–16
cutting text 4:25

D

debugger 6:1, 3, 5–6, 10; 7:2–3, 7, 10
debugging programs 3:78
debugging tools 6:1–2, 7
definition, argument 1:6
  command 1:6
  option 1:6
deleting mail message 5:8
deleting rest of buffer 4:32
deleting text 4:7, 21
determining command execution status 3:67
determining existing permissions 3:31
directives, compiler 6:1, 5, 9–10
directories 1:4, 8; 3:1–3, 5, 7–12, 14–15, 17–18, 26, 30–31, 34, 36, 57; 4:30–58

changing 3:2
creating 3:10
making 3:2, 10
naming 3:9, 11
organizing 3:10
protect 3:18, 30
removing 3:17
directory :ii; 1:4, 7; 2:3; 3:1–19, 25–33, 36, 41, 44, 47, 53, 56–58, 60, 62, 67–69, 74, 79–80, 83–84, 88, 92; 4:3; 5:3–4, 8, 10, 13–14, 16; 6:3; 7:3–18, 5, 7; A:2
changing current 3:15
child 3:8, 12, 16
current 3:4, 7–9, 11–13, 15, 27, 29–30, 36, 41, 53, 56–57, 74, 79; 5:17; 7:7
current working 3:4–7, 15; 5:10–16
home 2:3; 3:2–4, 10, 13, 15–18, 44, 57, 62, 84, 88, 92; 4:3; 5:4–4, 8, 10, 13; 7:5–14
login 3:3, 74
moving to 3:2
parent 3:7–8, 12, 17–18
directory permissions 3:15, 18, 30
print working 3:4, 92
root 3:3, 5, 9
source 3:3
discarding shell output 3:71
display differences between files 3:18, 36
documenting program functions 3:66
duplicate file 3:18, 25

E

editing multiple files 4:35
editing text with vi 4:6
editor, link 6:8
screen 3:88; 4:1, 6
vi :ii
entering X 2:1
environment, modifying login 3:84
errors, typing 1:9
Ethernet communication 5:13
exchanging messages 5:1
executable programs 1:6; 3:9, 62, 87–88
executing a simple shell program 3:56
executing command on remote system 5:12, 14
executing commands sequentially 3:44

exiting mail 5:8
exiting X 2:2

F

feature, alias 3:81, 84
  history 1:5; 3:81, 83
file, copy 3:18, 25, 92
  count characters 3:18, 29
  count lines 3:18, 29
  count words 3:18, 29
  duplicate 3:18, 25
  move 3:18, 26, 92
  ordinary 1:4; 3:2
file permissions 3:18, 30
  recovering lost 4:34
  remove 3:18, 28, 92
  special 1:4; 3:2, 15
file system 1:4; 3:1–8, 10–11, 16, 19, 27, 71; A:2
  viewing a 4:35
filenames, listing all 3:13
files, accessing 3:18
  archive 6:8
  combine object 6:8
  display differences 3:18, 36
  editing multiple 4:35
  manipulating 3:13, 15, 18
  naming 3:9
  ordinary 1:4; 3:2
  protect 3:18, 30
  special 1:4; 3:2
finding line number with line editor 4:31
finding out system name 5:6
fixing transposed letters 4:26
format, listing in short 3:14
formats, constructing multiple conditional 3:75
Fortran compiler options 6:11
full directory name 3:5
full pathname 3:5–6, 8–9, 12, 26–27, 44, 79; 5:10
functions, documenting program 3:66

G

global substitution 4:32
graphics tablet 1:3

H

hard copy 3:18
hardware, system interface 1:1–2
header, printing message 5:8
help, man online 1:6
history feature 1:5; 3:81, 83
home directory 2:3; 3:2–4, 10, 13, 15–18, 44, 57, 62, 84, 88, 92; 4:3; 5:4–4, 8, 10, 13; 7:5–14

I

iconifying windows 2:5
icons 1:3, 12; 2:2–14
in, logging :ii; 1:1, 10, 12; 3:4; 5:12–13
input, redirecting 3:46; 5:11
interpreter, command 1:5; 3:1, 39; 4:3
interrupting mail message 5:9

J

joining two lines 4:29
joy sticks 1:3
junction box 1:1, 3

K

kernel 1:4, 8
keyboard 1:1; 2:5; 3:46–3, 86; 4:7; 5:16; 7:5
knob box 1:3

L

.login 1:11–12, 15; 3:3–4, 15, 40, 47, 50–51, 54, 57, 59, 62, 66, 74, 83–84, 86, 89; 4:3; 5:1–9, 11–14, 17, 19 3:18, 26
language, C programming 1:5
C-shell programming 3:82, 89
shell command :ii; 3:40

leaving input mode 4:6
letters, fixing transposed 4:26
libraries, archive 6:1–2, 8, 10
library 3:88; 6:4, 8, 15; A:2
light pen 1:3
line editor commands 3:83; 4:30–84, 32–33
line editor commands in vi 4:30
  moving to 4:17
line numbers 4:17; 6:12, 15
lines, joining two 4:29
link editor 6:8
listing all filenames 3:13
listing directory contents 3:2, 11
listing in long format 3:14
listing in short format 3:14
literal characters 1:9; 3:45
loader table options 6:14
local to remote file copy 5:17
logging in :ii; 1:1, 10, 12; 3:4; 5:12–13
logging in to remote system 5:12–13
logging off 1:14; 3:55
login directory 3:3, 74
looping with /f2for/f1 loop 3:67
looping with /f2if...then...else/f1 loop 3:73
looping with /f2if...then/f1 loop 3:72
looping with /f2whilw -do/f1 loop 3:70
loops, conditional 3:72

M

mail :ii; 1:12; 2:2; 3:47, 49–51, 54, 59, 62, 83, 87; 5:1–12, 18; A:2–19
  exiting 5:8
  managing incoming 5:7
mail message commands 5:8
  undeliverable 5:4
making directories 3:2, 10
man online help 1:6
manager, window 7:11
managing incoming mail 5:7
manipulating files 3:13, 15, 18
matching, pattern 3:1, 29, 40–42
merge files 3:18, 38
message, deleting mail 5:8
  interrupting mail 5:9
  printing mail 5:8

resuming mail 5:9
saving mail 5:8
messages, exchanging 5:1
mode, leaving input 4:6
vi command 4:2
vi input 4:2
vi last 4:2
modem, remote communication with 5:12, 14
modifying login environment 3:84
modifying text 4:22
modifying windows 2:3
monitor 1:1, 3, 12
mouse 1:1, 3, 13; 2:2; 7:3–5, 5, 7
mouse pad 1:1, 3, 13; 2:4–14
mouse pointer 1:13; 2:4–5
move file 3:18, 26, 92
moving cursor 4:10–16
moving cursor line by line 4:12
moving cursor paragraph by paragraph 4:15
moving cursor sentence by sentence 4:14
moving cursor to first character of line 4:11
moving cursor to last character of line 4:11
moving cursor to specific character on line 4:12
moving cursor within window 4:16
moving cursor word by word 4:13
moving to home directory 3:2
moving to specified line 4:17

N

name, full directory 3:5
named variables 3:58, 61, 84, 86
naming directories 3:9, 11
naming files 3:9
naming shell programs 3:58
networking 5:12, 18; A:2
numbers, line 4:17; 6:12, 15

O

off, logging 1:14; 3:55
online commands 1:6
operating system :i, ii; 1:1, 3–8, 10–12, 14; 3:1, 3, 5, 9, 37–39, 44, 90; 4:34; A:1
optimization, code 6:1–2, 8, 10

compiler 6:11
option definition 1:6
options, C compiler 6:13
C preprocessor 6:14
command line 6:1, 3, 9
Fortran compiler 6:11
loader table 6:14
ordinary file 1:4; 3:2
ordinary files 1:4; 3:2
organizing directories 3:10
output, discarding shell 3:71
redirecting 3:23–24, 47–48, 50

P

.profile 3:13, 57, 84; 4:3; 6:7–4, 12, 14
pad, mouse 1:1, 3, 13; 2:4–14
page through file contents 3:18, 20
parent directory 3:7–8, 12, 17–18
password 1:11; 5:13–12, 17
pasting text 4:25
pathname 3:5–13, 15–17, 26–27, 44, 57, 69, 79, 88; 5:10, 13, 16
absolute 3:5
full 3:5–6, 8–9, 12, 26–27, 44, 79; 5:10
relative 3:7–9, 12–13, 16–17, 26–27
pattern matching 3:1, 29, 40–42
pattern matching a character set 3:42
pattern matching all characters 3:41
pattern matching one character 3:42
search file 3:19, 37, 92; 4:18
pen, light 1:3
permission, change directory 3:18, 30
change file 3:18, 30
permissions, changing 3:33, 35–36
determining existing 3:31
directory 3:15, 18, 30
file 3:18, 30
pointer, mouse 1:13; 2:4–5
positional parameter variables 3:58–59, 65
print contents on screen 3:18
print partially formatted file contents 3:18, 23
print working directory 3:4, 92
printer copy 3:18
printing mail message 5:8
printing message header 5:8

printing message number /f2n/f1 5:8
printing on screen 3:2
printing summary of mail commands 5:8
prints contents on screen 3:19
programming, shell 3:52, 56, 66
programs, debugging 3:78
  executable 1:6; 3:9, 62, 87–88
  naming shell 3:58
prompt, system 1:9, 11; 3:22, 89; 4:4; 5:3–5
prompting user for variable values 3:63–64
protect directories 3:18, 30
protect files 3:18, 30

Q

quitting mail 5:8

R

receiving files via mail command 5:11
recovering lost file 4:34
redefining commands 3:81, 84
redirecting input 3:46; 5:11
redirecting output 3:23–24, 47–48, 50
redrawing window 4:29
relative pathname 3:7–9, 12–13, 16–17, 26–27
remote communication 5:12
remote communication 5:15
remote communication with modem 5:12, 14
remote to local file copy 5:17
remove file 3:18, 28, 92
removing directories 3:17
rename file 3:18, 26
repeating last command 4:28
replacing text 4:22
report file's character count 3:18
reserved variables 3:61–62, 86
resize box 2:5
resuming mail message 5:9
return codes 3:56, 66–67, 75
root 2:2; 3:3–6, 9; 7:11
root directory 3:3, 5, 9
running background process after logging off 3:55
running commands at later time 3:52
running process status 3:54

S

saving changes with line editor 4:31
saving mail message 5:8
screen editor 3:88; 4:1, 6
  print contents 3:18
  printing on 3:2
scrolling text 4:16
search file for pattern 3:18–19, 37, 92; 4:18
sending files via mail command 5:11
sending mail to many people 5:5
sending messages to yourself 5:3
sequentially, executing commands 3:44
shell :ii; 1:4–8, 10; 2:2; 3:1, 39–40, 42–47, 49, 52, 55–67, 69–72, 74, 76–77, 79–82, 84; 4:1–89, 5, 9–10, 30; 5:14
  Bourne 1:5; 5:14
shell command language :ii; 3:40
shell program variables 3:58
shell programming 3:52, 56, 66
sort files 3:18, 38
source directory 3:3
special characters 1:9; 3:43–10, 45, 49, 54
special file 1:4; 3:2, 15
special files 1:4; 3:2
special parameter variables 3:59–60
speed, typing 1:10
statements, compilation control 6:1, 4, 9
  unconditional control 3:78
status, determining command execution 3:67
  running process 3:54
sticks, joy 1:3
substituting character string with one command 4:32
substituting text 4:23
substitution, command 3:51, 63, 65, 79
  global 4:32
syntax, command line 1:6–7
system, file 1:4; 3:1–8, 10–11, 16, 19, 27, 71; A:2
system interface hardware 1:1–2
  operating :i, ii; 1:1, 3–8, 10–12, 14; 3:1, 3, 5, 9, 37–39, 44, 90; 4:34; A:1
system prompt 1:9, 11; 3:22, 89; 4:4; 5:3–5
System, X Window :i, ii; 1:1, 3, 12; 2:1; A:2

T

tablet, graphics 1:3
terminal configuration 1:1; 4:5
testing loop conditions 3:74–75
testing loop conditions with return codes 3:75
text, adding 4:7–8, 10
  changing 4:24
  copying 4:26, 31
  cutting 4:25
  deleting 4:7, 21
  modifying 4:22
  pasting 4:25
  replacing 4:22
  scrolling 4:16
  substituting 4:23
tools, debugging 6:1–2, 7
trackballs 1:3
transposing characters 4:26
turning off meaning of blank space 3:45
turning off special character meanings 3:45
typing errors 1:9
typing speed 1:10

U

unconditional control statements 3:78
undeliverable mail 5:4
undoing last command 4:21
UNIX, Berkeley 4.3 1:3
UNIX V.3 :ii
user-named variables 3:61–62
using comments in shell program 3:66
using shell variables 3:86

V

V.3, UNIX :ii
variables, named 3:58, 61, 84, 86
  positional parameter 3:58–59, 65
  reserved 3:61–62, 86
  special parameter 3:59–60
  user-named 3:61–62
  using shell 3:86

vi :i, ii; 3:17, 37, 52, 62, 83, 85, 88; 4:0–9, 12–17, 22, 25, 27; 7:3–36, 5–7
vi command mode 4:2
  editing text 4:6
vi editor :ii
vi input mode 4:2
vi last line mode 4:2
viewing a file 4:35

W

window, background 2:4
  clearing 4:29
window manager 7:11
  moving cursor 4:16
  redrawing 4:29
windows, iconifying 2:5
  modifying 2:3
writing text to new file 4:31

X

.Xdefaults 2:3, 11
.xdesktop 2:3, 6–7, 11
X, entering 2:1
  exiting 2:2
X Window System :i, ii; 1:1, 3, 12; 2:1; A:2

Y

yourself, sending messages to 5:3