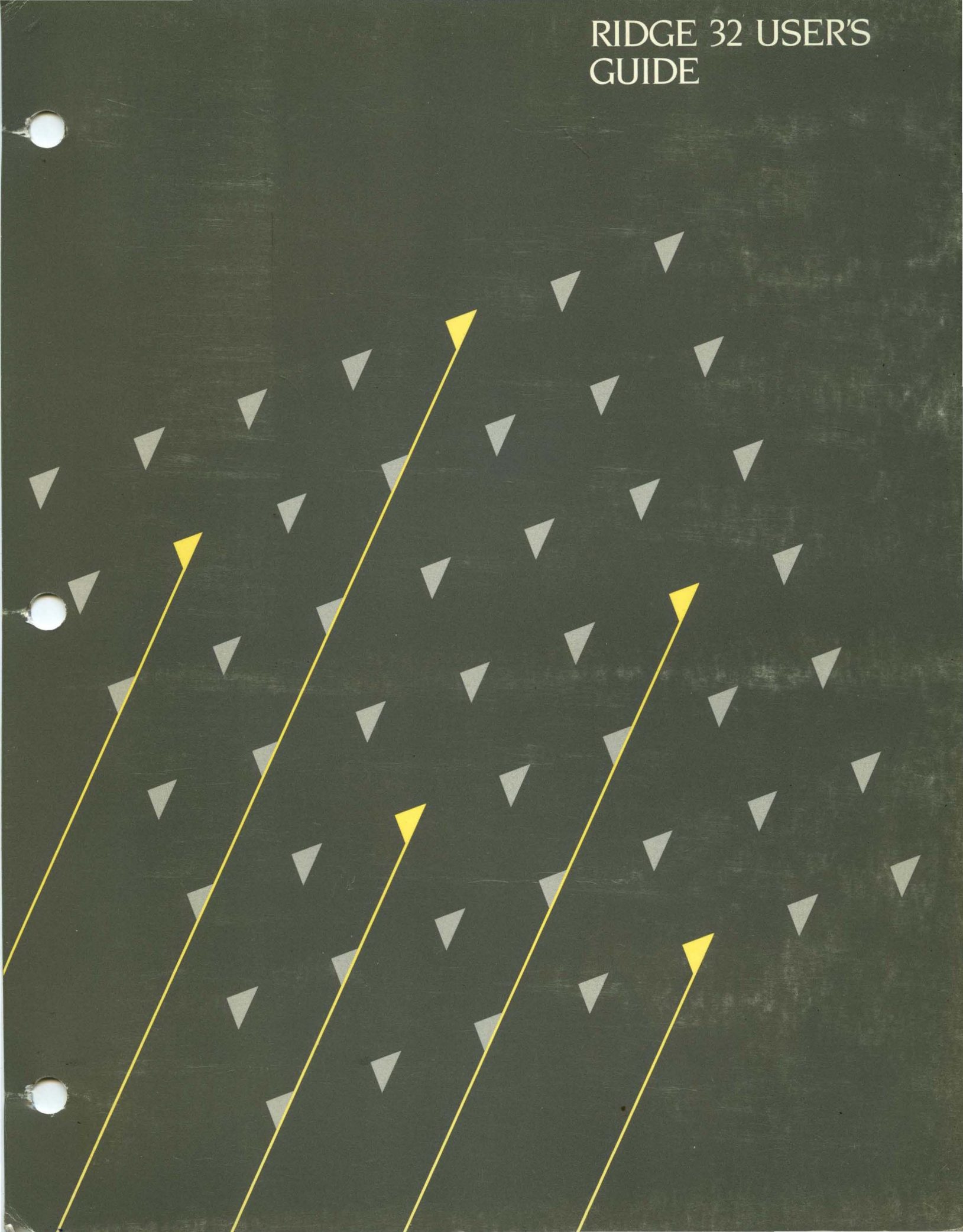
RIDGE 32 USER'S GUIDE



Ridge 32 User's Guide

First Edition: 9054 (OCTOBER 85)

PUBLICATION HISTORY

Manual Title: Ridge 32 User's Guide
First Edition: 9054 (OCTOBER 85)

NOTICE

FEDERAL COMMUNICATIONS COMMISSION NOTICE

This equipment generates and uses radio frequency energy and, if not installed and used properly, i.e. in strict accordance with the instructions manual, may cause harmful interference to radio communications. It has been tested and found to comply with the limits for a Class A computing device pursuant to subpart J of Part 15 of FCC rules, which are designed to provide reasonable protection against such interference when operated in a commercial environment. Operation of this equipment in a residential area is likely to cause interference, in which case the user may be required to correct the interference at his own expense.

ACKNOWLEDGEMENTS

This software and documentation is based in part on the fourth Berkeley Software Distribution, under license from the regents of the University of California. We acknowledge the following individuals for their part in its development: Ken Arnold, Rick Blau, Earl Cohen, Robert Corbett, John Foderaro, Mark Horton, Bill Joy, Howard Katseff, Randy King, Jim Kleckner, Steve Levine, Colin McMaster, Geoffrey Peck, Rob Pike, Eric Scott, and Eric Shienbrood.

Sections of the *Ridge User's Guide* are reprinted from the *UNIX™ System V User's Guide* under license from AT&T Bell Laboratories.

TRADEMARKS

PREFACE

This manual provides an overview of capabilities and technical procedures for the Ridge 32 product line, and refers the reader to the appropriate Ridge reference manuals.

Chapter 1 describes the Ridge 32 hardware, such as powering the system on and loading the disk and tape drives.

Chapter 2 provides an overview of the Ridge Operating System (ROS).

Chapter 3 describes how to log into ROS and some simple ROS commands.

Chapter 4 describes ROS command conventions and how to use the ROS file system.

Chapter 5 introduces the more advanced capabilities of the ROS system.

Chapter 6 is a tutorial on the use of the Bourne shell.

Chapter 7 describes the commands used to access the floppy disk drive

Chapter 8 describes the procedures to be performed by the System Administrator.

Chapter 9 describes how to care for your system hardware.

CONVENTIONS USED IN THIS MANUAL

An outline of a terminal display screen is used to set off examples of interactions between you and the ROS system. These examples apply regardless of the type of terminal you use. Inside the screen, the ROS system prompts and its responses are printed in `pica font`. The commands you type in response to the system prompts and your other input and data are printed in **bold**face type. These include the commands you type that do not appear on the screen (such as, a carriage return), which are enclosed in angle brackets < >. The following screen summarizes these conventions.

`pica`	(ROS system prompts and responses)
bold	(Your commands)
italics	(Variable which you or the computer substitutes a name or value)
<>	(Your commands or parts of commands that do not appear on the screen)
^g	(A control character, hold down the control key CTRL while your press "g".)

In the text, *italics* are used in the *command*(#) form to reference the number of the section in which a command is described in the **ROS Reference Manual.**

RIDGE 32 STANDARD MANUAL SET

This **User's Guide** refers to the following manuals which are shipped with every Ridge 32 system:

- **ROS Reference Manual** (order number 9010)
  This contains the reference page for each system command, utility, subroutine, library, and device file. Use this manual as the first source of information on any topic, then follow the references to the other manuals.
  The UNIX convention for referring to an entry in this manual is *commandname*(#), such as *cat*(1). The # is the numbered tab section of the **ROS Reference Manual** in which the *commandname* is explained. Other topics, which are not commands or programs, are referred to in the same form; example: *termio*(4) means that the "terminal i/o" topic is described in section four.

- **ROS Programmer's Guide** (order number 9050)
  This contains tutorial information on programming topics that are not covered in sufficient detail in the **ROS Reference Manual**. The **ROS Reference Manual** refers the reader to this manual when appropriate.

- **ROS Text Editing Guide** (order number 9051)
  This contains tutorial information on text-editing topics that are not covered in sufficient detail in the **ROS Reference Manual**. The **ROS Reference Manual** refers the reader to this manual when appropriate.

- **ROS Utility Guide** (order number 9053)
  This contains tutorial information on utility programs that are not covered in sufficient detail in the **ROS Reference Manual**. The **ROS Reference Manual** refers the reader to this manual when appropriate.

- **AT&T UNIX System V Graphics Guide** (order number 9026)
  This contains tutorial information on UNIX graphics programs that are not covered in sufficient detail in the **ROS Reference Manual**. The **ROS Reference Manual** refers the reader to this manual when appropriate.

- **Ridge Hardware Reference Manual** (order number 9007)
  Describes the theory of operation and programming of Ridge input/output boards, such as the disk and tape controllers.

- **Ridge Processor Reference Manual** (order number 9008)
  Describes the processor architecture and design criteria of the Ridge "reduced" instruction set.

Other Ridge manuals that describe optional peripherals and software are shipped with the product; you can also order most Ridge manuals from the product price list.

HOW TO ORDER MANUALS

Use the **Technical Publications Order Form** at the back of this book. Order a Ridge manual by its 4-digit document number (like **9004**) and its title. Based on the 4-digit document number, Ridge Order Processing will ensure you receive the most recent edition of a manual, the binder that comes with it, and any updates that exist at the time.

Manual editions are identified with sequential letters of the alphabet. The first edition has the 4-digit document number. Starting with the second edition, new editions have a new letter suffix. The suffixes start with **A** and continue alphabetically.

Manual update packages are identified with sequential numbers, starting with **1**.

Example of Manual Part Numbers

- The first edition of the XYZ manual may be number **9004**.
- The first update is **9004-1**.
- The second edition is **9004-A**.
- The first update to the second edition is **9004-A1**.
- The second update to the second edition is **9004-A2**.
- The third edition is **9004-B**.

CONTENTS

Chapter 1: THE RIDGE 32

Chapter 2: THE RIDGE OPERATING SYSTEM (ROS)

Chapter 3: GETTING STARTED

Chapter 4: USING THE FILE SYSTEM

Chapter 5: ROS SYSTEM CAPABILITIES

Chapter 6: SHELL TUTORIAL

Chapter 7: FLOPPY DISK DRIVE COMMANDS

Chapter 8: ADMINISTRATIVE DUTIES

Chapter 9: PREVENTIVE MAINTENANCE

LIST OF TABLES

LIST OF FIGURES

Chapter 1: THE RIDGE 32

Chapter 1
THE RIDGE 32

INTRODUCTION

The Ridge 32 is a minicomputer with mainframe power, integrated high-resolution graphics, and networking capabilities.

The Ridge 32 product line allows end users, system houses, and software developers to build integrated solutions for advanced computational applications. The large data storage capacity and high-speed computation capability of the Ridge 32 make it especially suitable for compute-intensive applications with large program files and data sets.

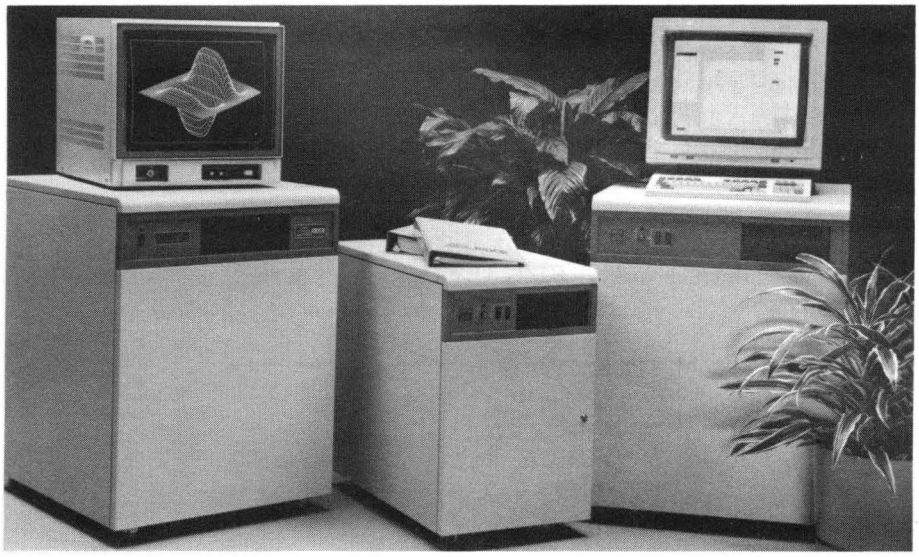


Figure 1-1. The Ridge 32 Product Line

INSTALLATION

Your Ridge 32 should be installed by a Ridge Systems Engineer. If you have not made arrangements to have your system installed, contact the Ridge Computers Customer Service department.

Do not attempt to install your Ridge 32 computer yourself.

TURNING THE SYSTEM ON

Once your Ridge 32 computer is installed, you can turn the power on by setting the power switch on the front panel to ON. After turning the power on, you will need access to the system console terminal, which is the terminal connected to the port marked **J1** (if an RS-232 terminal) or **Display1** (if a Bit-mapped terminal, such as a Ridge display). Within 45 seconds after turning on the power, the following appears on the system console's screen:

```
current date and time
Do you want to run with this date? (y)
```

If you wish to use the date and time displayed, press the RETURN key without an entry. To set a new date and time, enter **n** and press the RETURN key. The prompt **Enter new date (YYMMDDhhmm) :** will appear, after which you should type two-digit values for the year, month, day, hour, and minute in the **YYMMDDhhmm** form displayed by the prompt. Press the RETURN key.

For example, upon powering the system on, you notice that the date and time is set to May 6, 1985. 3:30pm, but the current date and time is May 18, 1985. 5:15pm. To set the correct date and time enter: **8505181715**. The screen will appear as follows:

```
May 6, 1985. 3:30pm
Do you want to run with this date? (y) n<CR>
Enter new date (YYMMDDhhmm): 8505181715<CR>
```

The BACKSPACE key does not work when entering the date in this way. Therefore, the date must be entered correctly on the first try. If you enter the date incorrectly, you can reset the date from the root account using the *date*(1) command. (See Chapter 8 for information on the root account.)

After a series of diagnostic messages, the **login:** prompt will appear, indicating the system is ready for use. You will find a description of the ROS operating system in Chapter 2 and instructions on how to log in and use some of the basic ROS commands in Chapter 3.

TURNING THE SYSTEM OFF

Only the person acting as the system administrator for your Ridge computer installation should turn the system power on or off. If a key is required to turn the system power off, it should be kept by the system administrator.

Do NOT turn off the power if:

- there are remote users
- the system is a link in a network
- jobs are scheduled to run when the system is unattended
- you are not sure what would be affected

Before turning off the power, you should use the **ps -a** command (see the *ps*(1) page in the *ROS Reference Manual*) to verify there are no shell or background processes running. After verifying that no user processes are running, it is safe to turn off the power if the `login:` prompt appears at all of the terminals connected to the system.

The Ridge 32 and peripheral equipment may be turned off when the system is not in use, but the electronics will not be harmed if they are left on overnight. If the system will not be used for more than a day, it may be wise to turn it off.

DISK DRIVES

The Ridge 32 has two types of disk drives:

- Hard Disk Drive
- Floppy Disk Drive

HARD DISK DRIVE

Your system is equipped with a Winchester hard disk drive. This means the hard disk is non-removable and should only be serviced by a Ridge Systems Engineer. Depending on which system you have selected, the hard disk will store between 78 and 445 Mbytes of data.

FLOPPY DISK DRIVE

The floppy disk drive is located at the top of the Ridge 32 computer on the right-hand side. This drive accepts double-sided, double-density, 8-inch floppy disks. Each formatted floppy disk can store up to 1.248 Mbytes of data.

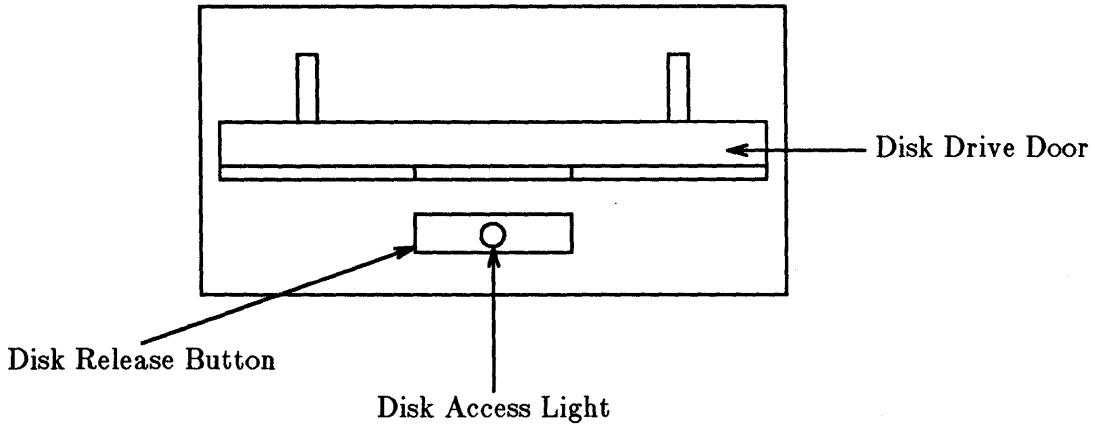


Figure 1-2. The Floppy Disk Drive Mechanism

Inserting a Floppy Disk

Before opening the door on the floppy disk drive, verify that the disk access light on the drive is off. If the disk access light is on, a disk is in the drive and is currently being read from or written to by the computer.

WARNING
NEVER REMOVE A FLOPPY DISK WHEN THE DISK ACCESS LIGHT IS ON!

When the disk access light is off, push the disk release button. If a disk is in the drive, it will be ejected.

Before inserting the floppy disk into the drive, see if the disk has a notch on the left side of the slotted end (see Figure 1-3). This is the "write disable" notch. If the write disable notch is exposed, you will only be able to read data from the disk. If you plan to write data to the disk, cover the write disable notch with one of the adhesive tabs packaged with your new disk.

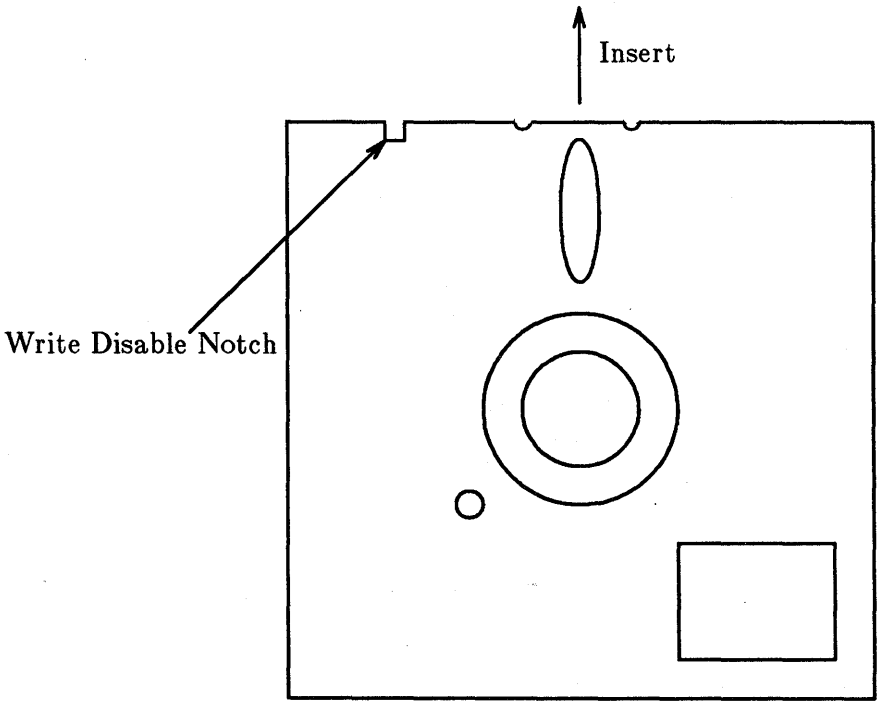


Figure 1-3. Floppy Disk

Insert the floppy disk, label side up and slotted edge first, all the way into the disk drive until you feel it "lock" into place. Pull down the disk drive door until it clicks shut.

The disk is now ready to be accessed by the floppy disk drive. See Chapter 7, *FLOPPY DISK DRIVE COMMANDS* for details on how to format and access the floppy disk.

TAPE DRIVE

A reel-to-reel, half-inch tape drive is available in the Ridge Companion Enclosure. The versatility of the tape drive software (see *mt*(7)) allows you to transfer data between the Ridge 32 and many types of computer systems.

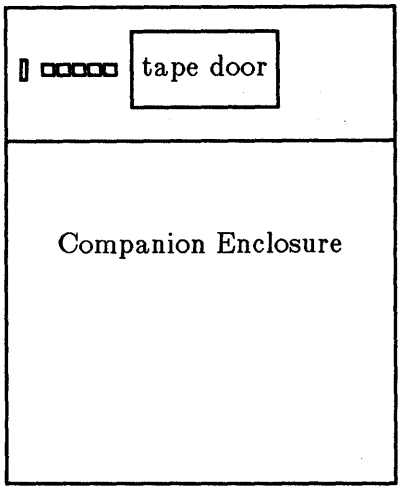


Figure 1-4. Position of the Tape Drive in the Companion Enclosure

CAUTION

Do not force open the plastic door of the tape drive when it is in use. The door opens easily by hand when the tape is not loaded.

Figure 1-5. Control Buttons on Front Panel of the Companion Enclosure

AUTOMATIC LOADING

The tape drive can accommodate tapes up to 10.5 inches in diameter. To load a tape onto the drive:

1. Open the front panel of the Companion Enclosure and press the power button on the lower right of the tape drive mechanism to the ON position. After an automatic and brief self-test cycle, the UNLOAD light should remain on.
2. By hand, wind the tape neatly and completely onto the reel to be loaded.
3. Press in at the bottom of the tape drive door to open it.
4. If the tape is to be written to, make sure the write-enable ring is fitted into the groove at the bottom of the tape spool. If the tape is to be read from only, you can protect the data on the tape from accidentally being overwritten by leaving the write-enable ring off. Insert the tape reel, label side up, into the opening.
5. Close the tape drive door.
6. Press the LOAD/REWIND button. The LOAD/REWIND light blinks during automatic threading of the tape. After successful loading, the LOAD/REWIND and WRT EN-TEST (if the write-enable ring is installed) lights stay on. If the LOAD/REWIND light blinks for more than one minute, automatic loading has failed. In this event power the tape drive off and back on, press UNLOAD, and repeat this step again. In the rare event that loading fails three times, manual loading is necessary.
7. When the LOAD/REWIND light stays on, press ON-LINE. Now the LOAD/REWIND, WRT EN-TEST (if write-enable ring is installed), and ON-LINE lights should be on. The tape is now ready to be accessed.

AUTOMATIC UNLOADING

To unload a tape automatically:

1. Take the tape drive off-line by pressing ON-LINE. The ON-LINE light should go off.
2. Press UNLOAD. As the tape is rewinding and unloading inside the tape drive, the UNLOAD light blinks. When the UNLOAD light stays on, the tape has finished unloading, and the door unlocks.
3. Open the tape drive door and remove the tape. Close the door.

MANUAL UNLOADING

In the rare event that the tape will not unload automatically, it must be removed from the drive by hand.

1. Remove the top lid of the Companion Enclosure by pulling up on the front edge of the lid, as pictured. (Some enclosures have quick release screws on the back of the cover. On these enclosures, unscrew the screw, pull the cover towards you, then lift up.)

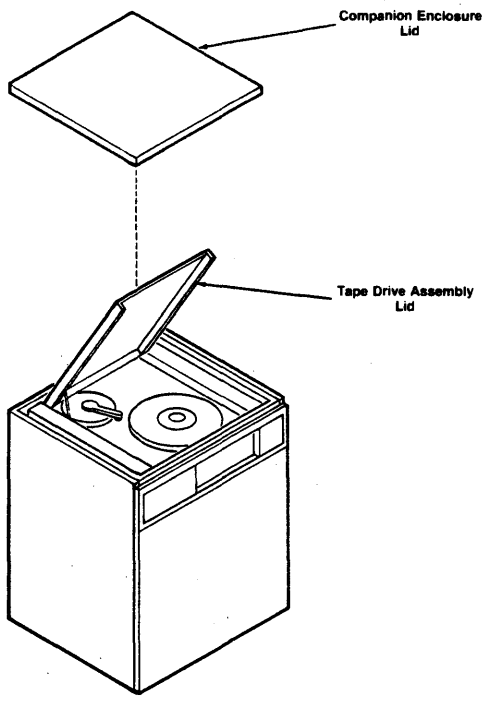


Figure 1-6. Accessing the Tape Drive Mechanism Through the Top of Companion Enclosure

2. Raise the lid of the tape drive mechanism. Prop it open with the plastic stick attached to the left side of the lid.
3. Release the supply reel by pressing and holding the white button underneath it, then turning the supply reel by hand until all the tape is wound onto it.
4. Turn the supply reel until it resists, then turn it past the point of resistance. The drive will release the reel.
5. Close the lid and put the Companion Enclosure back in the original operating position.

Chapter 2: THE RIDGE OPERATING SYSTEM (ROS)

Chapter 2
THE RIDGE OPERATING SYSTEM

INTRODUCTION

The Ridge Operating System (ROS) is derived from two popular variants of the UNIX operating system: UNIX System V from AT&T Bell Laboratories and the 4.2 Berkeley Software Distribution (bsd) from the University of California at Berkeley. If you are familiar with a UNIX system, you already have a working knowledge of ROS.

ROS software does three things:

- It controls the computer
- It acts as an interpreter between you and the computer
- It provides a package of programs or tools that allow you to do your work

The ROS software that controls the computer is referred to as the operating system. The operating system coordinates all the details of the computer's internals, such as allocating system resources and making the computer available for general purposes. The nucleus of this operating system is called the kernel.

The ROS software that acts as a liaison between you and the computer is called the shell. The shell interprets your requests and, if valid, retrieves programs from the computer's memory and executes them.

The ROS software that allows you to do your work includes programs and packages of programs for electronic communication, for creating and changing text, and for writing programs and developing software tools.

This package of services and utilities called ROS offers:

- A *general purpose* system that makes the resources and capabilities of the computer available to you for performing a wide variety of jobs or applications, not simply one or a few specific tasks.
- A computing environment that allows for an *interactive* method of operation so you can directly communicate with the computer and receive an immediate response to your request or message.
- A technique for sharing what the system has to offer with other users, even though you have the impression that ROS is giving you its undivided attention. This is called *timesharing.* ROS creates this feeling by allowing you and other users--*multiusers*--slots of computing time measured in fractions of seconds. The rapidity and effectiveness with which ROS switches from working with you to working with

other users makes it appear that the system is working with all users simultaneously.

- A system that provides you with the capability of executing more than one program simultaneously, this feature is called *multitasking.*

HOW ROS WORKS

This section describes the main components of the Ridge Operating System. Look at Figure 2-1. It shows a set of layered circles in graduated sizes. Each circle represents specific ROS software, such as:

- Kernel
- Shell
- Programs/tools that run on command

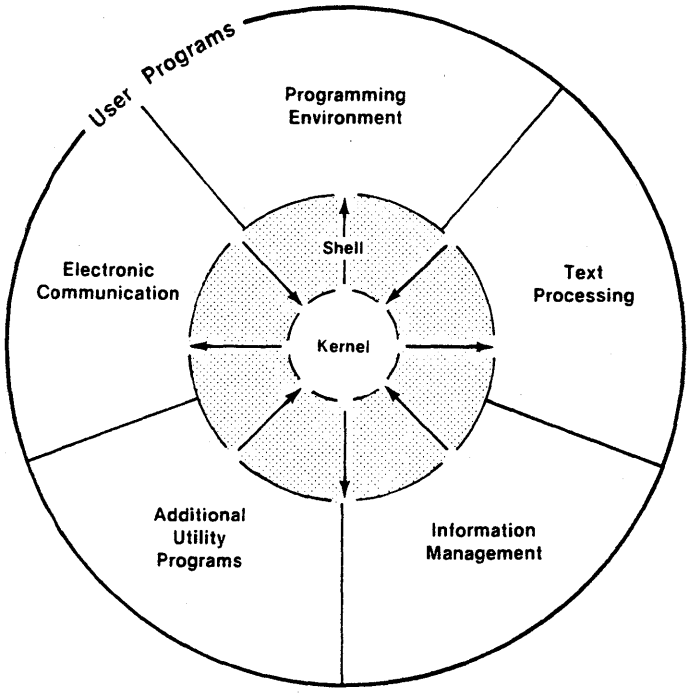


Figure 2-1. ROS System Model

You should know something about the major components of ROS software to communicate with the ROS system. Therefore, the remainder of this chapter introduces you to each component: the kernel, the shell, and user programs or commands.

KERNEL

The heart of ROS is called the kernel. Figure 2-2 gives an overview of the kernel's activities. Essentially, the kernel is software that controls access to the computer, manages the computer's memory, and allocates the computer's resources to one user, then to another. From your point of view, the kernel performs these tasks automatically. The details of how the kernel accomplishes this are hidden from you. This arrangement lets you focus on your work, not on the computer's.

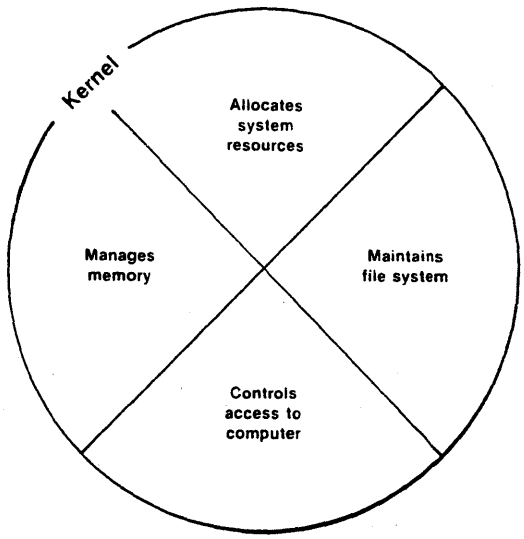


Figure 2-2. Functional View of Kernel

On the other hand, you will become increasingly familiar with another feature of the kernel; this feature is referred to as the file system.

File System

The file system is the cornerstone of the Ridge Operating System. It provides you with a logical, straightforward way to organize, retrieve, and manage information electronically. If it were possible to see this file system, it might look like an inverted tree or organization chart made up of various types of files.

The file is the basic unit of ROS and it can be any one of three types:

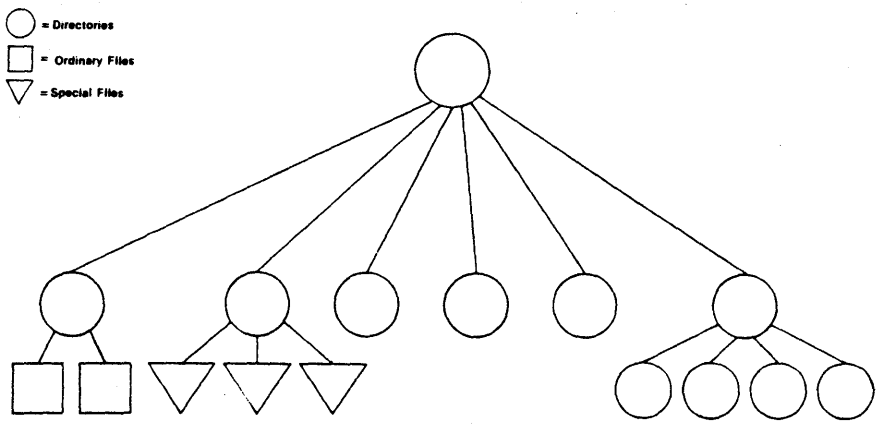


Figure 2-3. Treelike ROS Directory Structure

- An *ordinary file* is simply a collection of characters. Ordinary files are used to store information. They may contain text or data for the letters or reports you type, code for the programs you write, or commands to run your programs. In ROS, everything you wish to save must be written into a file.

  In other words, a file is a place for you to put information for safekeeping until you need to recall or use its contents again. You can add material to or delete material from a file once you have created it, or you can remove it entirely when the file is no longer needed.

- A *directory* is a file maintained by the operating system for organizing the treelike structure of the file system. A directory contains files and other directories as designated by you. You can build a directory to hold or organize your files on the basis of some similarity or criterion, such as subject or type.

  For example, a directory might hold files containing memos and reports you write pertaining to a specific project or client. Or a directory might hold files containing research specifications and programming source code for product development. A directory might hold files of executable code allowing you to run your computing jobs. Or a directory might contain files representing any combination of these possibilities.

- A *special file* represents a physical device, such as the terminal on which you do your computing work or a disk on which ordinary files are stored. At least one special file corresponds to each physical device supported by ROS.

In some operating systems, you must define the kind of file you will be working with and then use it in a specified way. You must consider how the files are stored since they can be sequential, random-access, or binary files. To ROS, however, all files are alike. This makes the ROS file structure easy to use. For example, you need not specify memory requirements for your files since the system automatically does this for you. Or if you or a program you write needs to access a certain device, such as a printer, you specify the device just as you would another one of your files. In ROS, there is only one interface for all input from you and output to you; this simplifies your interaction with the system.

The source of the ROS file structure is a directory known as root, which is designated with a slash (/). All files and directories in the file system are arranged in a hierarchy under root. Root normally contains the kernel as well as links to several important system directories that are shown in Figure 2-4:

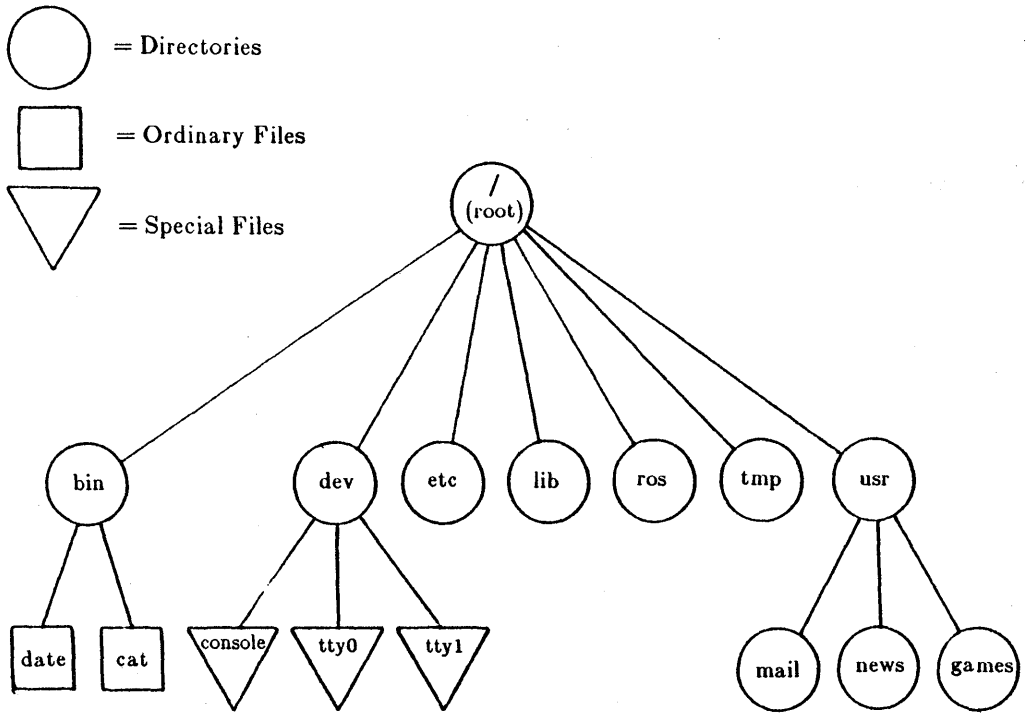


Figure 2-4. Typical File System Structure

/bin Many executable programs and utilities reside in this directory.

/dev This directory contains special files that represent peripheral devices, such as the console, the line printer, user terminals, and disks.

/etc Programs and data files for system administration can be found in this directory.

/lib This directory contains available program and language libraries.

/ros This directory contains ROS system files. These files should not be changed or executed from the command line.

/tmp This directory is a place where anyone can create temporary files.

/usr This directory holds other directories, such as **mail** (which further holds files storing electronic mail), **news** (which contains files holding newsworthy items), and **games** (which contains files holding electronic games).

In summary, the directories and files you create comprise the portion of the file system that is structured and, for the most part, controlled by you. Other parts of the file system are provided and maintained by the operating system, such as **bin, dev, etc, lib, tmp** and **usr**, and have much the same structure on all UNIX and ROS systems.

Chapter 4 shows how to organize a file system directory structure and how to access and manipulate files. Chapter 5 gives an overview of ROS capabilities. The effective use of these capabilities depends on your familiarity with the file system and your ability to access information stored within it.

SHELL

The shell is a unique ROS program or tool that is central to most of your interactions with the ROS system. Figure 2-1 illustrates how the shell works. The drawing shows the shell as a circle containing arrows pointing away from the kernel and the file system to the outer circle that contains programs and then back again. The arrows indicate that a two-way flow of communication is possible between you and the computer via the shell.

When you enter a request to the ROS system by typing on the terminal keyboard, the shell translates your request into language the computer understands. If your request is valid, the computer honors it and carries out an instruction or set of instructions. Because of its job as translator, the shell is called the command language interpreter.

As the command language interpreter, the shell can also help you to manage information. The shell's ability to manage information stems from the design of the ROS system. Each ROS program is designed to do one thing well. In a sense, a ROS program is a building block or module that you can use in tandem with other programs to create even more powerful tools.

In addition to acting as a command language interpreter, the shell is a programming language complete with variables and control flow capabilities.

ROS allows you to use two types of shells:

- Bourne Shell
- C-Shell

In order to keep the discussions of shell use as simple as possible, this guide will only describe the operations of the Bourne shell. The C-shell offers some special features which you may find desirable. For information on the use of the C-shell, see the *C-Shell* section in the *ROS Programmer's Guide* and the *csh*(1) pages in the *ROS Reference Manual.*

The *WORKING IN THE SHELL* section in Chapter 5 describes the Bourne shell's capabilities. The tutorial in Chapter 6 teaches you how to use these capabilities to write simple shell programs called shell scripts and how to custom-tailor your computing environment.

Commands

A program is a set of instructions that the computer follows to do a specific job. In ROS, programs that can be executed by the computer without need for translation are called executable programs or commands.

As a typical user of the ROS system, you have many standard programs and tools available to you. If you also use the ROS system to write programs and to design and develop software, you have system calls, subroutines, and other tools at your disposal. And you have, of course, the programs you write.

This book introduces you to approximately 40 of the most frequently used programs and tools that you will probably use on a regular basis when you interact with the ROS system. If you need additional information on these or other standard ROS system programs, check the *ROS Reference Manual.* If you want to use tools and routines that relate to programming and software development, you should consult the *ROS Programmer's Guide* and the *ROS Utility Guide.*

The details contained in the *ROS Reference Manual* may also be available via your terminal in what is called the *on-line* version of the *ROS Reference Manual.* This on-line version is made up of formatted text files that look like the printed pages in the manuals. You can summon pages in this electronic manual using the command **man**, which stands for **man**ual page. If the electronic version of the manuals is available on your computer, the **man** command is documented in the *man*(1) page of your copy of the *ROS Reference Manual.*

What Commands Do

The outer circle of Figure 2-1 organizes ROS system programs and tools into general categories according to what they do. The programs and tools allow you to:

- *Process text.* This capability includes programs, such as, line and screen editors (which create and change text), a spelling checker (which locates spelling errors), and optional text formatters (which produce high-quality paper copies that are suitable for publication).

- *Manage information.* The ROS system provides many programs that allow you to create, organize, and remove files and directories.

- *Communicate electronically.* Several programs, such as **mail**, provide you with the capability to transmit information to other users and to other ROS systems. See the *ROS Utility Guide* for deatils on the mail utility.

- *Use a productive programming and software development environment.* A number of ROS system programs establish a friendly programming environment by providing ROS system-to-programming-language interfaces and by supplying numerous utility programs.

- *Take advantage of additional system capabilities.* These programs include graphics, a desk calculator package, and computer games.

How Commands Execute

Figure 2-5 gives a general idea of what happens when the ROS system executes a command.

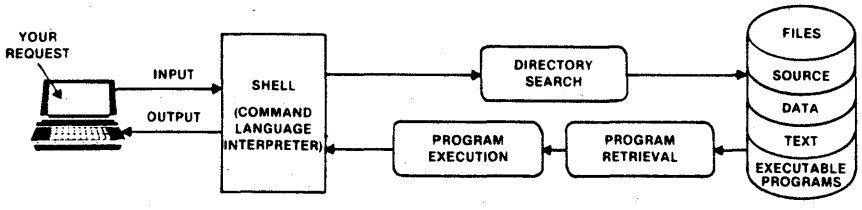


Figure 2-5. Flow of Control During Program Execution

When the shell signals it is ready to accept your request, you type in the command you wish to execute on the keyboard. The command is considered input, and the shell searches one or more directories to locate the program you specified. When the program is found, the shell brings your request to the attention of the kernel. The kernel then follows the program's instructions and executes your request. After the program runs, the shell asks you for more information or tells you it is ready for your next command.

This is how the ROS system works when your request is in a format that the shell understands. The structure that the shell understands is called a command line. Chapter 4 explains what you need to know about the command line so you can request a program to run.

This chapter has outlined some basic principles of the ROS operating system and explained how they work. The following chapters will help you begin to apply these principles according to your computing needs.

Chapter 3: GETTING STARTED

Chapter 3
GETTING STARTED

INTRODUCTION

There are general rules and guidelines with which you should be familiar before you begin to work on the ROS system. For example, you need information about your terminal and how to use its keyboard and about how to begin and end a computing session.

This chapter acquaints you with these rules and guidelines and presents you with information to help to make your first encounter with the ROS system understandable and to lay the groundwork for future computing sessions. Since the best way to learn about the ROS system is to use it, this chapter helps to get you started by providing examples of how to use these rules and guidelines to establish contact with the ROS system and to respond to its requests and prompts.

To establish contact with the ROS system, you need:

- A terminal
- An identification name, called a login name, by which the ROS system recognizes you as one of its authorized users
- A password with which the ROS system double-checks and verifies your identity after you log in and before it allows you to use its resources

ABOUT THE TERMINAL

A terminal is an input/output device: through it you input a request to the ROS system and the system, in turn, outputs a response to you. The terminal is equipped with a keyboard, a monitor or display unit, a control unit, and a link that allows it to communicate with the computer.

The terminal you use to interact with the ROS system can be either a Ridge Display (Figure 3-1) or any type of RS-232 terminal.

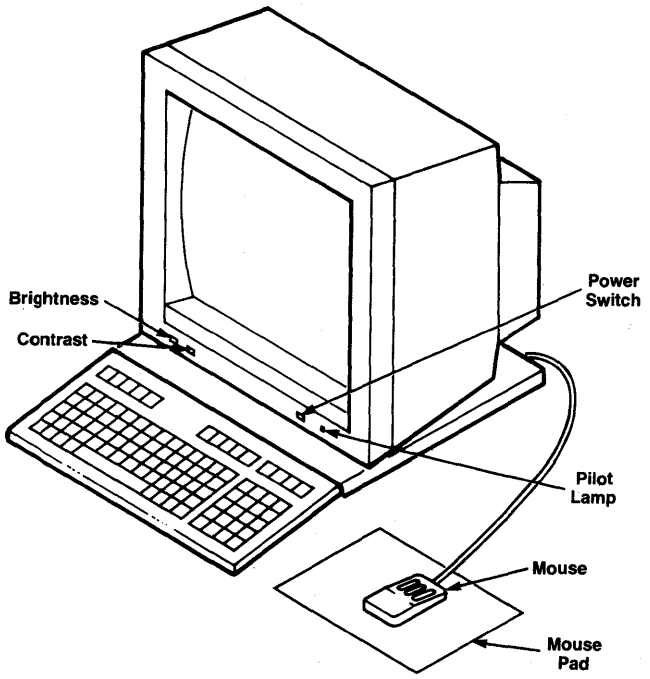


Figure 3-1. Ridge Graphics Display Terminal

REQUIRED TERMINAL SETTINGS

Regardless of the type of terminal you use, you must set it up or configure it in a certain way to ensure proper communication with the ROS system. If you have not configured a terminal before, you might feel more comfortable seeking help from your System Administrator.

How you configure a terminal depends on the type of terminal that you are using. Some terminals are configured with switches, whereas other terminals are configured directly from the keyboard using a set of function keys. To determine how to configure your terminal, consult the owner's manual provided by the manufacturer. If you are connecting an RS-232 terminal, the *ADDING AN RS-232 TERMINAL* section in Chapter 8 provides the general guidelines for connecting an RS-232 terminal. If you are connecting a Ridge Graphics Display, follow the installation procedure outlined in the user's guide shipped with the display.

KEYBOARD CHARACTERISTICS

Figure 3-2 illustrates the keyboard layout of the Ridge Graphics Display Terminal.

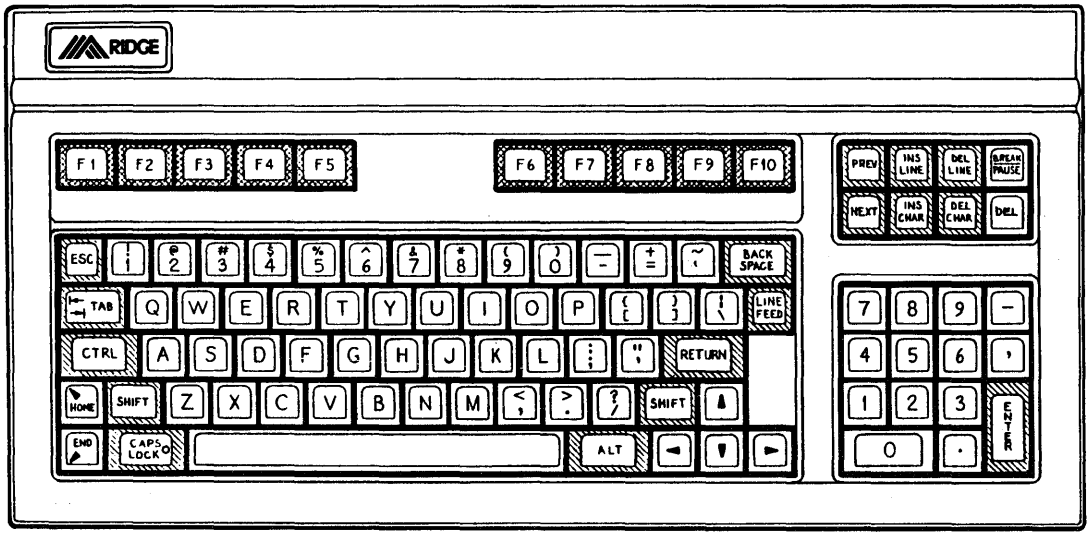


Figure 3-2. Ridge Graphics Display Keyboard Layout

Its keys correspond to:

- Letters of the English alphabet **a** through **z** and **A** through **Z** when you are holding down a shift key,
- Numeric characters 0 through 9,
- A variety of symbols, such as ! @ # $ % ^ & ( ) _ – + = ~ ‘ { } [ ] \ : ; " ’ < > , ? /
- Words, such as RETURN and BREAK, and abbreviations, such as DEL (delete), CTRL (control), and ESC (escape).
- Special Function keys. Many terminals have special keys that can be assigned a particular function by software.

Many of the keys corresponding to symbols, words, and abbreviations have been added to the keyboard layout and the placement of these characters or symbols on a keyboard may vary from terminal to terminal.

Consequently, there is not a truly standard layout for terminal keyboard characters. There is, however, a standard set of characters that keyboards have, consisting of 128 characters, called the ASCII character set. ASCII is the abbreviation for American

Standard Code for Information Interchange. When you depress a key or combination of keys, the appropriate ASCII code is sent to the computer for translation from the alphabetic and numeric characters that we understand to electronic signals that the computer can decode.

After you configure the terminal and survey its keyboard, all you need is a login name to establish communication with the ROS system.

OBTAINING A LOGIN NAME

When you attempt to establish contact with the ROS system, ROS must be able to verify that you are an authorized user. You identify yourself as an authorized user by means of a login name.

To receive a login name, set up a ROS system account through your local System Administrator or the person in charge of your Ridge 32 computer installation.

Your login name is determined by local practices. Possible examples are your last name, your nickname, or a ROS system account number. Typically, a login name is three to eight characters in length. It can contain any combination of alphanumeric characters, as long as it starts with a letter. It cannot, however, contain any symbols. The following are examples of legal ROS system login names: **starship**, **mary2**, and **jmrs**.

ESTABLISHING CONTACT WITH THE ROS SYSTEM

When the system's power is on and startup sequence described in Chapter 1 is complete, your terminal should display the `login:` prompt.

If you are greeted with a series of meaningless characters, depress the BREAK key. If the ROS system does not display the `login:` prompt within a few seconds, depress the BREAK key once again.

LOGIN PROCEDURE

When the connection is made and the ROS system prompts for your login name, type in your login name and depress the **<CR>** key. In the following examples, **starship** is the login name.

```
login: starship<CR>
```

Remember to type your login name in the same case as it was established. By convention, login names are established in lowercase letters.

PASSWORD

If you do not have a password yet, the system will display a message, such as:

```
login: starship<CR>
Welcome to the Ridge - This is ROS 3.3
$
```

If the system did not request a password, you should assign one to your account by typing

```
login: starship<CR>
Welcome to the Ridge - This is ROS 3.3
$ passwd<CR>
```

Your password should be as least seven characters long and can be made up of any combination of alphabetic and numeric characters. Examples of valid passwords are: **mar84ch**, **Jonath0n**, and **BRAV3S**.

Enter the name you wish to use as your password after the `new password:` prompt. Notice your password entry does not appear on your screen as you type it. This is to ensure that only you know the password to your account. After pressing the **<CR>** key, you will be asked to retype your password to confirm that you entered it correctly the first time. Re-enter your password and press the **<CR>** key. (The **passwd** command is discussed in the *passwd*(1) pages of the *ROS Reference Manual.*)

The next time you log in, ROS will ask you for both your user name and your password, as demonstrated by the user **starship** in the following example:

```
login: starship<CR>
password:your_password<CR>

Welcome to the Ridge - This is ROS 3.3

$
```

If you made a typing mistake that you did not correct before depressing <**CR**>, the ROS system displays the message `login incorrect` on your terminal monitor and asks you to try again by printing the `login:` prompt.

```
login: ttarship<CR>
password:your_password<CR>
login incorrect
login:
```

POSSIBLE PROBLEMS WHEN LOGGING IN

A terminal usually behaves predictably providing you have configured it properly. Sometimes, however, it may act peculiarly. For example, each character you type may appear twice on the terminal monitor or the carriage return may not work properly.

Some problems can be corrected by entering **stty sane**. Note that when you enter this command, it may not be echoed back to your display. If this command doesn't work, try reentering it and, instead of pressing the <**CR**> key, press the line feed key. If all else fails, log out and back in again. If none of these actions remedy the problem, you should check the following and try logging in once again:

- *Keyboard*--Keys that are marked CAPS, LOCAL, BLOCK, and so on should not be enabled, that is, in the locked position. You can usually disable these keys simply by depressing them.
- *Data phone set or modem*--If your terminal is connected to the computer via telephone lines, verify that the baud rate and duplex settings are correctly specified.
- *Switches*--Some terminals have several switches that must be set to be compatible with the ROS system. If this is the case with the terminal you are using, make sure they are set properly.

If all of the users connected to the same system have terminal problems, you might have to reboot the system (see Chapter 8). It is never necessary to turn the Ridge 32 off and on again.

Table 3-1 presents a list of procedures you can follow to detect, diagnose, and correct some problems you may experience when trying to establish contact with the ROS system. If none of the possibilities covered in the table helps you, contact the system administrator or the person in charge of the Ridge 32 computer installation at your location.

Table 3-1
Troubleshooting Problems When Logging in*

Problem†	Possible Cause	Action/Remedy
Stream of meaningless characters when logging in	ROS system attempting to communicate at wrong speed	Depress BREAK key
Input and output is printed in uppercase letters	Terminal configuration includes UPPERCASE setting	Log off, set character generation to LOWERCASE, and log in again
Input is printed in UPPERCASE letters, output in LOWERCASE	Key marked CAPS or CAPS LOCK is locked or enabled	Depress the CAPS or CAPS LOCK key to disable setting
Input is printed (echoed) twice	Terminal is set to HALF DUPLEX mode	Change setting to FULL DUPLEX mode
Tab key does not work properly	Tabs are not set to advance to next	Type stty -tabs‡
Communication link cannot be established in spite of receiving high pitched tone when dialing in	Terminal is set to LOCAL or OFF-LINE mode	Set terminal to ON-LINE operation and try logging in again
Communication link between terminal and ROS system is repeatedly dropped on logging in	Terminal is set to LOCAL or OFF-LINE mode	Call system administrator

* Numerous problems can occur if your terminal is not configured properly. To eliminate these possibilities before attempting to log in, perform the configuration checks listed on page 2-4.

† Some problems may be specific to your terminal, data set, or modem, check the owner's manual for this equipment if suggested actions do not remedy the problem.

‡ Typing **stty -tabs** corrects tab setting only for your current computing session. To insure correct tab setting for all sessions, add the line **stty -tabs** to your .profile (see Chapter 8).

TYPING CONVENTIONS

To interact effectively with the ROS system, you should be familiar with certain typing conventions. An example of a ROS system typing convention is using lowercase letters when you issue commands. Other typing conventions require that you use combinations of characters to erase letters and delete lines, temporarily stop the ROS system from sending output to your terminal, and so on.

The next few pages introduce you to these conventions. Table 3-2 lists these special characters, keystrokes, and their meanings for quick reference.

Table 3-2
ROS System Typing Conventions

Key(s)	Meaning
DEL	Stop execution of a program or command
ESC	Use with another character to perform specific function (called escape sequence) Or, use to indicate end of create mode when using screen editor (**vi**)
RETURN	End a line of typing; designated as <**CR**>
BACKSPACE	Erase a character
Control x	Erase an entire line
Control d	Stop input to system or log off; designated as <^**d**>
Control h	Erase a character for terminals without a BACKSPACE key; designated as <^**h**>
Control i	Horizontal tab for terminals without a tab key; designated as <^**i**>
Control s	Temporarily stops output from printing on screen; designated as <^**s**>
Control q	Resumes printing after typing <^**s**>; designated as <^**q**>

NOTE: All control characters are sent by holding down the control key and pressing the appropriate letter.

Responding to the Command Prompt

The ROS system command prompt will be either $, %, or #, depending on whether you are using the Bourne shell, C-shell, or are logged in as **root**. When the system prompt appears on your terminal monitor, it means that the ROS system is waiting for you to enter a command and press the carriage return key, designated as <**CR**> throughout this guide, to execute it.

The $, %, and # are the default command prompts. Chapter 6 explains how to change the default value to another prompt.

Type-Ahead Buffer

ROS features a type-ahead buffer that allows you to type characters while the system is computing previous input or writing on the screen. Up to 256 characters can be typed before the system prompt appears and the system is ready to accept input.

Normally, if you type a command before the system prompt appears, the command will not be echoed (or displayed) on your terminal screen until the system completes its current task. If you wish your input to be displayed the moment you type it, enter:

```
$ stty echoi
```

When you log off the system, type-ahead will be reset to its original echo state. See the **echoi** mode for the **stty** command in the *stty*(1) pages of the *ROS Reference Manual* for details.

Stopping a Command

If you wish to stop the execution of a command, simply depress the DEL key. In turn, you will receive the system prompt indicating that the ROS system terminated the running of the program and is ready to accept your next command.

Control Characters

Locate the control key on your terminal keyboard. The key may be labeled CTRL or CONTROL and is probably to the left of the A key or below the Z key. The control character is used in combination with other keyboard characters to initiate a physical controlling action across a line of typing, such as backspacing or tabbing. In addition, some control characters define ROS-specific commands, such as temporarily halting output from printing on a terminal monitor.

Type a control character by holding down the CTRL key and depressing an appropriate alphabetic key. Control characters do not print on the terminal when typed. In this guide, control characters are designated with a preceding carat (^), such as <^**s**> for control s, to help identify them.

The following sections describe some of the control character combinations you will be using regularly when working with the ROS system.

Correcting Typing Errors

You can correct typing errors in two ways providing you have not pressed <**CR**>. The **BACKSPACE** key allows you to erase previously typed characters on a line, and <^**x**> allows you to delete the entire line on which you are working.

Temporarily Stopping Output

At times, you may wish to stop the ROS system temporarily from printing output on your terminal monitor. This is useful when you wish to keep information from rolling off the screen monitor on a video display terminal. If you type <^**s**>, printing is suspended; typing <^**q**> causes the printing to resume.

Terminating a Computing Session

When you have completed a session with the ROS system, you should type <^**d**>. This is the recommended way to log off the system and is described in detail later in this chapter.

Additional Control Character Capabilities

The ROS system furnishes other control character capabilities. For instance, if your terminal keyboard does not have a backspace key, typing <^**h**> gives you a backspace. Typing <^**i**> gives you a tab key if your terminal is set properly. (Refer to the section entitled *Possible Problems When Logging In* for information on how to set the tab key.)

SIMPLE COMMANDS

When the system prompt is displayed on your monitor, the ROS system recognizes you as an authorized user. Your response to the system prompt is to request ROS system programs to run.

Type in the command **date** and press <**CR**> after the command prompt. When you do this, the ROS system retrieves the **date** program and executes it. As a result, your terminal monitor should look something like the following.

```
$ date<CR>
Wed Oct 12 09:49:44 PST 1983
$
```

As you can see, the ROS system prints the date and the time. In this example, the PST stands for Pacific Standard Time. Your terminal monitor will display the appropriate time for your geographical location.

Now type the command **who** and depress <**CR**>. Your screen will look something like this.

```
$ who<CR>
starship   tty00    Oct 12   8:53
mary2      tty02    Oct 12   8:56
acct123    tty05    Oct 12   8:54
jmrs       tty06    Oct 12   8:56
$
```

The **who** command lists the login names of everyone currently working on your system. The `tty##` designations refer to the names of the special files that correspond to the terminals on which you and other users are currently working. The login date and time for each are also given.

LOGGING OFF

When you have completed a session with the ROS system, you should type <^**d**> after the system prompt. (Remember that control characters such as the <^**d**> are typed by holding down the control key and depressing the appropriate alphabetic key.) Since they are nonprinting characters, they do not appear on the terminal monitor. In a few seconds, the ROS system should display the `login:` message again. This indicates you have logged off successfully and someone else can log in at this time. Your terminal monitor should look like the one that follows.

```
$ <^d>
login:
```

It is strongly recommended that you log off the system using <^**d**> before turning off the terminal or hanging up the phone. It is the only way to assure you have been logged off the ROS system.

Chapter 4: USING THE FILE SYSTEM

Chapter 4
USING THE FILE SYSTEM

INTRODUCTION

To use the file system effectively you must be familiar with its structure, know something about your relationship to this structure, and understand how the relationship changes as you move around within the file system. Reading this chapter serves as preparation to use this file system.

The first ten or so pages should help to give you a working perspective of the file system. These pages contain information on the makeup of the file system and on how you fit into its organization. The remainder of the chapter introduces you to a number of ROS system commands. Some of these commands allow you to build your own directory structure, whereas others allow you to access and manipulate the subdirectories and files you organize within it. There are also commands that allow you to examine the contents of other directories in the system that you have permission to look at or to use.

Each command is discussed in a separate subsection in a way that will allow you to use it effectively. Many of the commands presented in this section have additional, sophisticated uses; these, however, are left for more experienced users and are described in other ROS system documentation. You can read these sections in the order in which they are presented in the text or you can read about the commands and their capabilities in the order that best suits your interests and purpose. Nevertheless, all the commands presented are basic to using the file system efficiently and easily. It is recommended that you read through them thoroughly and then try them out. Before viewing how the file system is structured, however, let's take a look at the structure of a command.

COMMAND STRUCTURE

For the ROS system to understand your intentions when using commands, you must take care to see that you input commands using the correct format, called the command line syntax. The command line syntax provides a procedure for ordering elements in a command line. It serves the same purpose as putting words in a certain sequence or order so that you can meaningfully express your ideas and thoughts to others. Without sentence structure, people would have difficulty interpreting what you mean. Similarly, without command line syntax, the ROS system shell cannot interpret your request.

Command line syntax consists of one or more of the following elements separated by a blank or blanks and followed by pressing the carriage return <**CR**> key:

command option(s) argument(s)

where

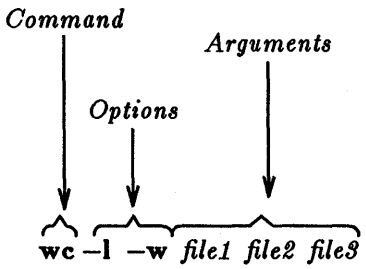
command is the name of the program you wish to run

option modifies how the command runs

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

A command line can simply contain a command name followed by <**CR**>, or it can list options and/or arguments in addition to the command. If you specify options and arguments on the command line, you must separate them with at least one blank. Blanks can be typed by pressing the space bar or the tab key. If a blank is part of the argument name, enclose the argument in double quotation marks, for example, "sample 1".

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** have been specified. (The ROS 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
wcfile	wc file
wc–lfile	wc –l file
wc –l w file	wc –lw file *or* wc –l –w file
wc file1file2	wc file1 file2

You can refer back to the ground rules on command line syntax as you read and work through the chapter.

HOW THE FILE SYSTEM IS STRUCTURED

The file system comprises a set of directories, ordinary files, and special files. These components provide you with a way to organize, retrieve, and manage information. Chapter 2 introduced you to directories and files, but let's review what they are before learning how to use them to tap the resources of the file system.

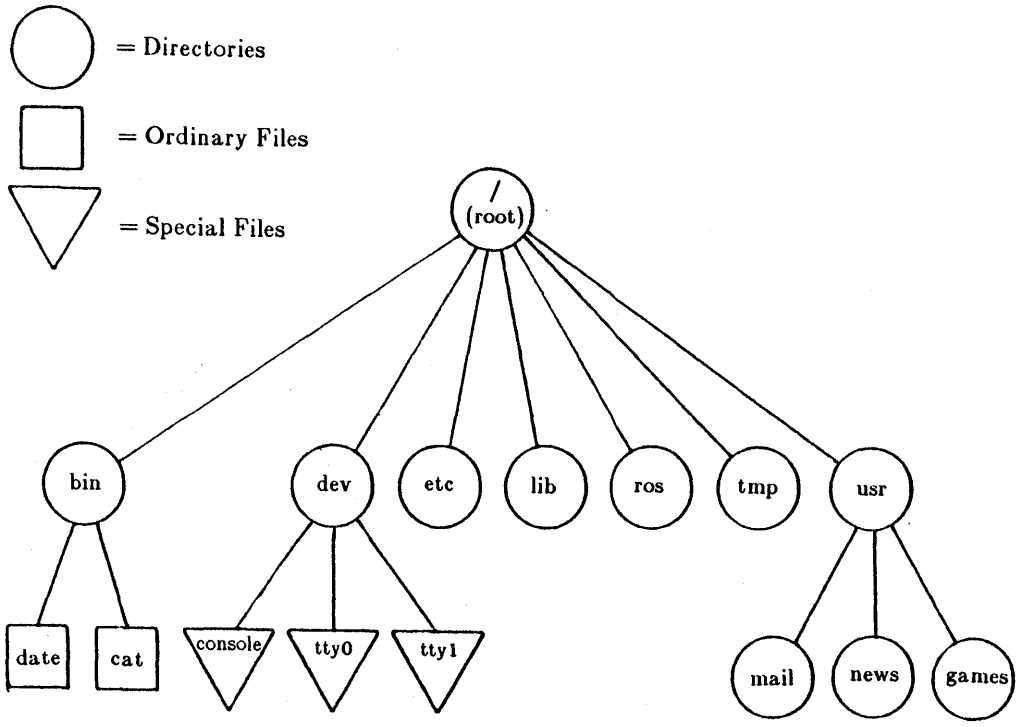


Figure 4-1. Sample File System

In general, a directory is a collection of files and other directories. Specifically, it contains the names of these files and directories. You can build a directory to organize the files you create on the basis of some similarity. An ordinary file is a collection of characters stored on a disk. Such a file may contain text for a status report you type or code for a program you write. Any information you wish to save must be written into a file. And a special file represents a physical device, such as your terminal.

The set of all the directories and files is organized into a treelike structure. Figure 4-1 helps you to visualize this. It shows a single directory called **root** as the source of a sample file structure. By descending the branches that extend from root, several other major system directories can be reached. 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 that are subordinate to a directory have what is called a parent/child relationship. This type of relationship is possible for many generations of files and directories, giving you the capability to organize your files in a variety of ways.

YOUR PLACE IN THE FILE SYSTEM STRUCTURE

When you are interacting with the ROS system, you will be doing so from a location in its file system structure. The ROS 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 the directories and files you own and to access those belonging to others that you have permission to use.

The following sections describe your place in relation to the file system structure and how this relationship changes as you move through the file system.

YOUR HOME DIRECTORY

When you successfully complete the login procedure, the ROS system positions you at a specific point in its file system structure called your login or home directory. The login name that was assigned to you when your ROS system account was set up is usually the name of this home directory. Every user with an authorized login name has a unique home directory in the file system.

The ROS system is able to keep track of all these home directories by maintaining one or more system directories that organize them. For example, let's say that the name of one of these system directories is **user1**, and that it contains the home directories of the login names **starship**, **mary2**, and **jmrs**. Figure 4-2 shows you how a system directory like **user1** ranks in relation to the other important ROS system directories you read about in Chapter 2.

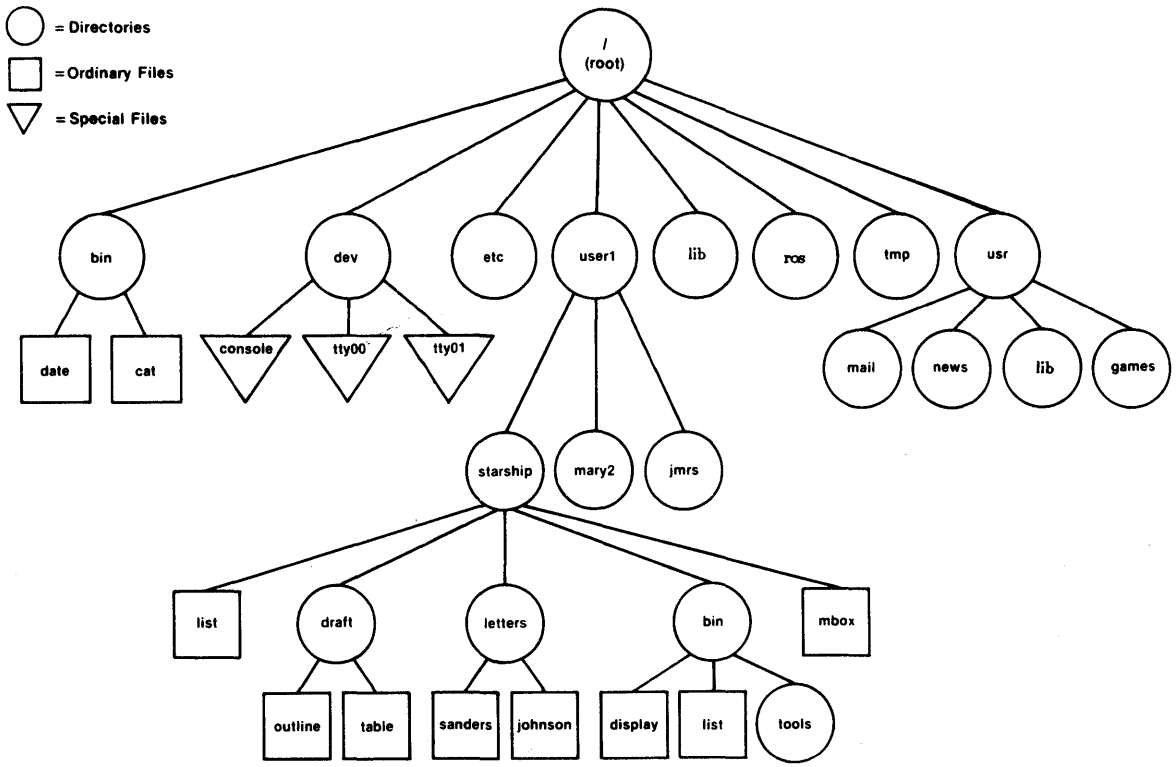


Figure 4-2. User and System Directories in the ROS File System

Within your home directory, you can create files and additional directories (sometimes called subdirectories) to organize them, you can move and delete these files and directories, and you can control who can access your files and directories. You have full responsibility for everything you create in your home directory because you own it. Your home directory is a vantage point from which to view all the files and directories it holds. It is also a point from which to view the file system all the way up to root.

YOUR WORKING DIRECTORY

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

There is a ROS system command called **pwd**, which stands for **p**rint **w**orking **d**irectory, that you can use to verify the name of the directory in which you are currently working.

For example, if your login name is **starship** and you issue the **pwd** command in response to the first $ prompt after logging in, the ROS system should respond as follows:

```
$ pwd<CR>
/user1/starship
$
```

The system reply indicates that your working directory is **/user1/starship**. Technically, **/user1/starship** is the full or complete name of your working directory, which indicates your current location in the ROS file system. Strings like **/user1/starship** are referred to as a path names. Path names are discussed in the next section.

We will analyze and trace this path name in the next few pages so you can start to move around in the file system. For now, it is sufficient to say that what **/user1/starship** tells you is that the root directory / (indicated by the leading slash in the line) contains the directory **user1**, which in turn contains the current working directory, which is **starship**. All other slashes in the path name are simply used to separate names of directories and files.

Remember, you can always use the **pwd** command to determine where you are in the file system. The **pwd** command will be especially helpful if you try to read or copy a file and the ROS system tells you that the file you are trying to access does not exist. You may be surprised to find that you are in a different directory than you thought.

To provide you with a quick summary of what you can expect the **pwd** command to do, a recap of how to use it follows.

Command Recap

pwd - print full name of working directory

command	*options*	*arguments*
pwd	none	none
Description:	**pwd** prints the full path name of the directory in which you are currently working.	
Remarks:	If the system responds with messages, such as, **cannot open directory** or **read error in directory**, there may be problems with the file system. Inform the system administrator.	

PATH NAMES

Every file and directory in the ROS system is identified by a unique path name. The path name tracks or indicates the location of the file or directory in the file system structure. Additionally, a path name provides directions to a specific file or directory. Knowing how to follow the directions the path name gives is your key to moving around the directory structure successfully.

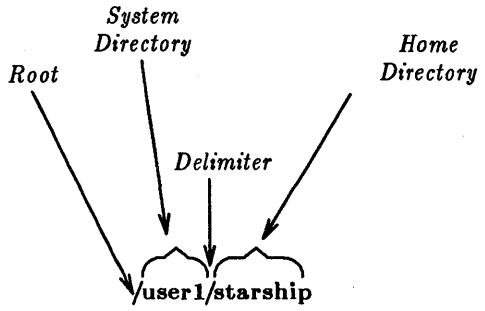
In the file system, there are two types of path names--full and relative.

Full Path Names

A full path name (sometimes called an absolute path name) gives you directions that take you from the root directory down through a unique sequence of directories that leads to a particular directory or file. You can use a full path name to reach any file or directory in the ROS system in which you are working. A full path name always starts at the root of the file system and its leading character is a / (slash). The final name in a full path name can be either a file name or a directory name. All other names in the path must be directories.

To understand how a full path name is constructed and where it can lead you, let's use the sample file system (Figure 4-2) and say that your current working directory is **starship**. If you issue the **pwd** command, the system responds by printing the full path name of your working directory, which is **/user1/starship**.

We can analyze the elements of this path name using the following diagram.



where:

/ (leading)	= Root of the file system when it is the first character in the path name,
user1	= System directory one level below root in the hierarchy to which root points or branches,
/ (subsequent)	= Slash that separates or delimits the directory names, **user1** and **starship**, and
starship	= Current working directory, which is also the home directory.

Now look at Figure 4-3, it traces the full path to **/user1/starship** through the sample file system we are using.

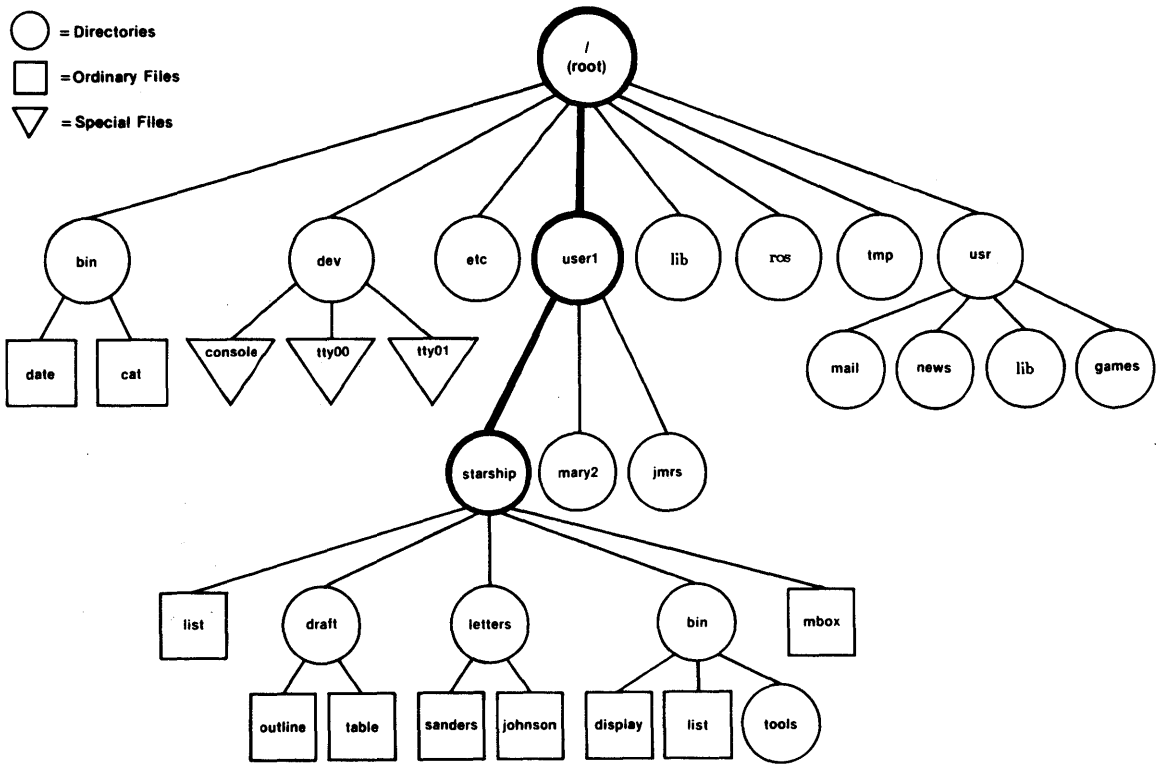


Figure 4-3. Full Pathname to the /user1/starship Directory Traced by Heavy Bold Lines

Relative Path Names

A relative path name is the name of a file or directory that varies with relation to the directory in which you are currently working. From your working directory, you can move "down" in the file system structure to access files and directories you own or you can move "up" in the hierarchy through generations of parent directories to the grandparent of all system directories, the root. A relative path name begins with a directory or file name, with a . (dot), which is a shorthand notation for the directory in which you are currently located, or a .. (dot dot), which is a shorthand notation for the directory immediately above your current working directory in the file system hierarchy. The .. (dot dot) is called the parent directory of the one in which you are currently located, which is the current directory or . (dot).

For example, if you are in the home directory **starship** in the sample system and **starship** contains directories named **draft**, **letters**, and **bin** and a file named **mbox**, the relative path name to any of these is simply its name, be it **draft**, **letters**, **bin**, or **mbox**. Figure 4-4 traces the relative path name from **starship** to **draft**.

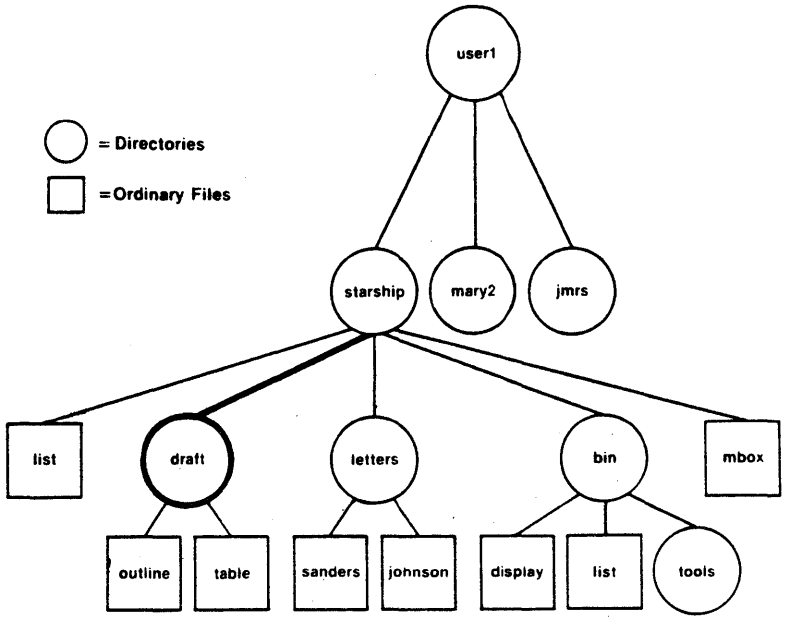


Figure 4-4. Relative Path Name for the Draft Directory is Traced With Heavy Bold Lines

Now, let's say the **draft** directory belonging to **starship** contains the files **outline** and **table**. Then, the relative path name from **starship** to the file **outline** is written as **draft/outline**.

Figure 4-5 traces this relative path. Notice that the slash in this path name separates the directory named **draft** from the file named **outline**. Here, the slash is a delimiter that indicates that **outline** is subordinate to **draft**; that is, **outline** is a child of its parent, **draft**.

Thus far, the discussion of relative path names covered how to specify names and directories of files that belong to, or are children of, your current directory -- in other words, to descend 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 your working directory, you can use the .. notation. This means that if you are in the directory named **draft** in the sample file system, .. is the path name to **starship**, and **../..** is the path name to **starship**'s parent directory **user1**. From **draft**, you could also trace a path to the file **sanders** in the sample system by using the path name **../letters/sanders** (.. brings you up to **starship**, then down to **letters**, and finally **sanders**).

Keep in mind that you can always use a full path name in place of a relative one.

In summary, some examples of full and relative path names would be:

Path Name	Meaning
/	Full path name of the root directory for the file system.
/bin	Full path name of the **bin** directory that contains most executable programs and utilities.
/user1/starship/bin/tools	Full path name of the directory called **tools** belonging to the directory **bin** that belongs to the directory **starship** belonging to **user1** that belongs to **root**.
bin/tools	Relative path name to the file or directory **tools** in the directory **bin**. If the current directory is /, then the ROS system searches for **/bin/tools**. But, if the current directory is **starship**, then the system searches the full path **/user1/starship/bin/tools**.
tools	Relative path name of a file or directory **tools** in the working directory.

It might take some practice to move around in the file system with confidence. But this is to be expected when learning a new concept and the techniques to use it.

To give you a chance to try your hand at moving about in the system's structure, the remainder of the chapter introduces you to the ROS system commands that make it possible for you to peruse the file system. If you lose track of where you are in the system's structure, use the **pwd** command to identify your location.

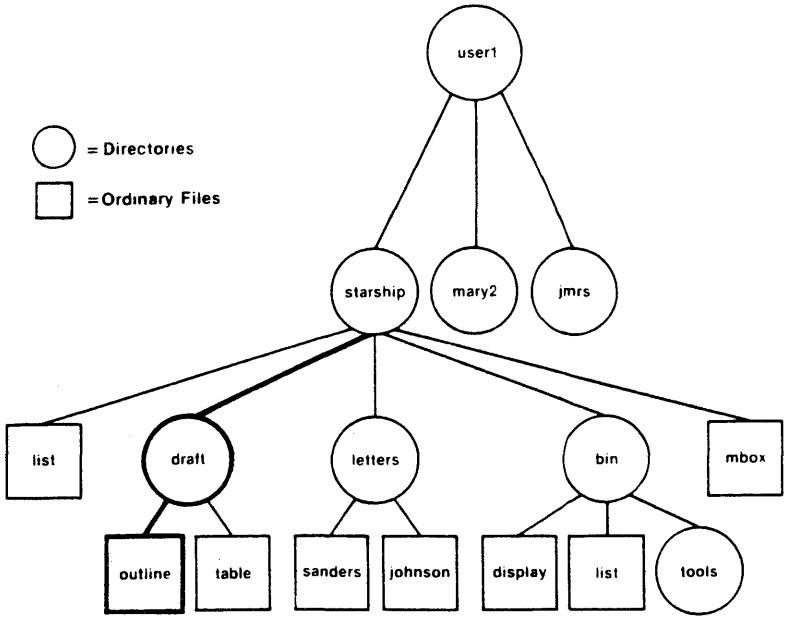


Figure 4-5. The Relative Path draft/outline is Traced in Bold Lines

ORGANIZING A DIRECTORY STRUCTURE

This section introduces you to four ROS system commands that make it possible for you to organize and use a directory structure. These commands and what you can expect them to do are as follows:

mkdir -- Allows you to create or make new directories and subdirectories from your current working directory

ls -- Allows you to list the names of all the subdirectories and files in a directory

cd -- Provides you with the ability to change your location from one directory to another in the file system

rmdir -- Lets you remove a directory when you no longer have a need for it

All of the commands can be used with path names--full or relative--when organizing a directory structure and when moving to the directories and subdirectories you organize, as well as when navigating to directories in the file system that belong to others that you have permission to access. The **ls** and **cd** commands can also be used without a path name.

Each of the commands is described more fully in the four sections that follow. In addition, a summary called a command recap is given for each command. The command recaps allow you to review quickly the command line syntax and the capabilities of each command.

CREATING DIRECTORIES *(mkdir)*

It is recommended that you create subdirectories in your home directory according to some logical and meaningful scheme to help you retrieve information you will keep in files. A convenient way to organize your files is to put all files pertaining to one subject together in a directory.

To create a directory, the ROS system provides you with the **mkdir** command, which stands for **mk**e **dir**ectory. In the sample file system, the **draft** subdirectory in the home directory **starship**, for example, may have been built by entering the following while located in **starship**:

```
$ mkdir draft<CR>
$
```

The $ response to the **mkdir** command indicates that a directory named **draft** was successfully created.

Similarly, the other subdirectories named **letters** and **bin** were created with the same command, as indicated in the following screen:

```
$ mkdir letters<CR>
$ mkdir bin<CR>
$
```

All the subdirectories (**draft, letters, bin**) could have been created in one command with the same results, as the following screen shows:

```
$ mkdir draft letters bin<CR>
$
```

You can also move to a subdirectory you created and build additional directories. When you build directories, or create files, keep in mind the following guidelines:

- The name of a directory (or file) can be from one to sixteen characters in length.
- All characters other than / are legal.
- Some characters are best avoided, such as a blank or space, a tab, or a 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.
- Avoid using the +, — or . as the first character in names.
- Uppercase and lowercase characters are distinct to the ROS system. For example, the directory or file named **draft** would not be the same as the directory or file named **DRAFT** or **Draft**.

Examples of legal directory or file names would be:

memo	MEMO	section2	ref:list
file.c	chap3+4	item1-10	outline

See the command recap that follows for a quick reference to **mkdir**'s capabilities.

Command Recap

mkdir - make a new directory

command	*options*	*arguments*
mkdir	none	**directoryname(s)**
Description:	**mkdir** creates a new directory (subdirectory).	
Remarks:	The system returns the $ prompt if the directory is successfully created.	

LISTING THE CONTENTS OF A DIRECTORY *(ls)*

All directories in the file system have information about the files and directories they contain, such as name, size, and the date last modified. You can obtain this information about what your working directory and other system directories contain by using the **ls** command.

The **ls** command, which stands for **list**, lists the names of the files and subdirectories of the directory you specify by path name. If you do not specify a path name, **ls** lists the names of files and directories in your working directory. To demonstrate how the **ls** command works, let's use the sample file system (Figure 4-2) once again.

You are logged into the ROS system and the shell responds to your **pwd** command with the line **/user1/starship**. To display the names of files and directories in the working directory, you would type **ls<CR>**. After this sequence, your terminal should read:

```
$ pwd<CR>
$ /user1/starship
$ ls<CR>
bin    draft    letters    list    mbox
$
```

As you can see, the system responds by listing the names of files and directories in the working directory **starship** in alphabetical order. If the first character of any of the file or directory names was a number or a capital letter, it would have been printed first.

Now, if you want to print the names of files and subdirectories in a directory other than your working directory without moving from your working directory, you should use the command format:

ls *directoryname<CR>*

where the *directoryname* is the full or relative path name of the desired directory. This means that you can print the contents of **draft** while you are working in **starship** by entering **ls draft<CR>**.

```
$ ls draft<CR>
outline    table
$
```

In the previous example, **draft** was a relative path name from **starship** to **draft**. Similarly, you could print the contents of the **user1** directory, which is the parent of the **starship** by typing:

```
$ ls ..<CR>
jmrs    mary2    starship
$
```

where .. is the relative path name from **starship** to **user1**. You could also list the contents of **user1** by typing **ls /user1<CR>** (since **/user1** is the full path name from root to **user1**) and get the identical listing.

In general, you can list the contents of any system directory that you have permission to access using the **ls** command and a full or relative path name.

The **ls** command is particularly useful if you have a long list of files and you are trying to determine whether one of them exists in your working directory. For example, if you are in the directory **draft** and you wish to determine if the files named **outline** and **notes** are there, you can use the **ls** command as follows:

```
$ ls outline notes<CR>
notes not found
outline
$
```

The output on the terminal monitor shows that the system acknowledges the existence of **outline** by printing its name, but says that the file **notes** is not found.

The **ls** command will not print the contents of a file. If you wish to see what a file contains, you can use the **cat**, **pg**, or **pr** command, which are described in the section of this chapter entitled *Accessing and Manipulating Files.*

Frequently Used *ls* Options

The **ls** command also accepts options that cause specific attributes of a file or subdirectory to be listed. There are more than a dozen available options for the **ls** commands. Of these, the **–a** and **–l** will probably be most valuable in your basic use of the ROS system. Refer to the *ROS Reference Manual* for information and details on the other options.

Listing All Names in a File. Some important file names in your home directory begin with a . (dot), such as .profile, . (the current directory), and .. (the parent directory). The **ls** command will not print these names unless you use the **–a** option in the command line. Thus, to list all files in your working directory **starship**, including those that start with a . (dot), type **ls –a<CR>**. The terminal should look something like this:

```
$ ls –a<CR>
.           draft       mbox
..          letters
.profile    list
$
```

Listing Contents in Long Format. Probably the most informative **ls** option is **–l**. If you type **ls –l<CR>** while in the **starship** directory, you would get the following:

```
$ ls –l<CR>
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
$
```

After the command line, the first line of output, **total 30**, shows the amount of memory used, which is measured in chunks called blocks. Next is one line for each directory and file. The first character in each of these lines tells you what kind of file is listed, where:

d = Directory,

- = Ordinary disk file,

b = Block special file, and

c = Character special file.

The next several characters, which are either letters or hyphens, describe who has permission to read and use the file or directory. (Permissions are discussed with the **chmod** command in the section entitled *Accessing and Manipulating Files* in this chapter.) The following number is the link count, which in the case of a file, equals the number of directories it is in, or in the case of a directory, also includes the number of directories immediately under it in the file system structure. Next is the login name of the owner of the file, which is **starship**, and then the group name of the file or directory, which is **project**. The following number indicates the length of the file or directory entry measured in units of information (or memory) called bytes. Then there is the month, day, and time that the file was last modified. Finally, the file or directory name is given.

Figure 4-6 sums up what you get when you list the contents of a directory in long format.

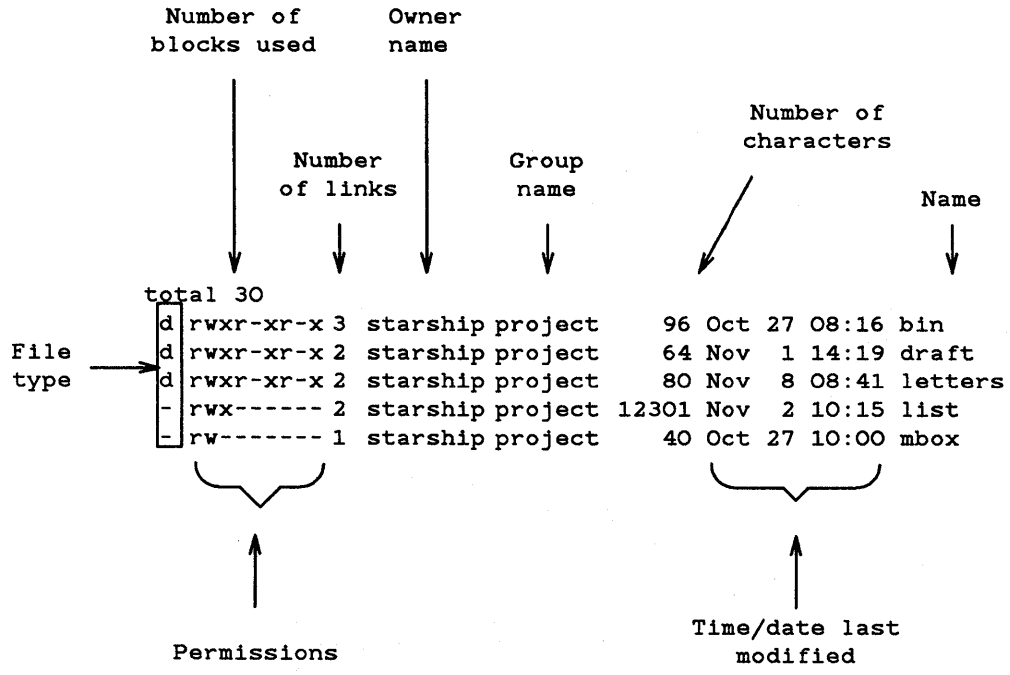


Figure 4-6. Output Produced by the ls –l Command

Command Summary. Following is a recap of capabilities provided by the **ls** command and two available options. See the *ROS Reference Manual* for information on other available options.

Command Recap

ls - list contents of a directory

command	*options*	*arguments*
ls	**–a, –l,** and others*	**directoryname(s)**
Description:	**ls** lists the names of the files and subdirectories in the specified directories. If no directory name is given as an argument, the contents of your working directory are listed.	
Options:	**–a** Lists all entries, including those beginning with . (dot).	
	–l Lists contents of a directory in long format furnishing mode, permissions, size in bytes, and time of last modification.	
Remarks:	If you want to read the contents of a file, use the **cat** command.	

* See the *ROS Reference Manual* for all available options and an explanation of their capabilities.

CHANGING YOUR WORKING DIRECTORY *(cd)*

When you first log into the ROS system, you are placed in your home directory, which becomes your current or working directory. You may, however, wish to work in a different directory for any number of reasons. For example, you might want to create a file in a specific directory, you may need to make corrections to a file in another directory, or you may wish to obtain information by reading a file in a different directory.

Whatever the reason, the ROS system provides you with the **cd** command that allows you to move around in its directory structure. When you use the **cd** command to move to a new directory, that directory becomes your working directory.

To use the **cd** command, enter the command:

cd *newdirectory-pathname*<CR>

where the path name, whether full or relative, to the new directory is optional. Any valid path name of a directory can be used as an argument to the **cd** command. If you use the **cd** command without specifying a path name, it will move you to your home directory regardless of where you are in the file system.

When you specify a valid directory path name on the command line, the ROS system moves you to that directory. For example, to move from the **starship** directory to the child directory **draft** in the sample file system, type **cd draft<CR>**. In this example, **draft** is the relative path name to the desired directory. When you get the $ prompt, verify your new location by typing **pwd<CR>**. Your terminal monitor should look something like the following:

```
$ cd draft<CR>
$ pwd<CR>
/user1/starship/draft
$
```

Now that you are in the **draft** directory you can access the files and directories in it, in this case, the files **outline** and **table**. You can also create subdirectories in **draft** with **mkdir** and additional files with the **ed** and **vi** commands. (See the *ROS Text Editing Guide* for general information on the **ed** and **vi** commands.

You may also use full path names with the **cd** command. For example, to move to the **letters** directory from the **draft** directory, you could use the command

cd /user1/starship/letters<CR>

where **/user1/starship/letters** is the full path name to **letters**.

Or, since **letters** and **draft** are both children of **starship**, you could use the **cd** command with the relative path name **../letters**. The **..** notation moves you to the directory **starship**, and the remainder of the path name moves you to **letters**.

If you wish to return to your home directory after perusing the file system, simply type **cd<CR>**. The **cd** command with no arguments returns you to your home directory.

Command Recap

cd - change your working directory

command	*options*	*arguments*
cd	none	**directoryname**

Description: **cd** changes your position in the file system from the current directory to the directory specified. If no directory name is given as an argument, the **cd** command places you in your home directory.

Remarks: When the shell places you in the directory specified, the $ prompt is returned to you. You will also receive a $ prompt when you issue the **cd** command with no argument. To access a directory that is not in your working directory, you must substitute the full or relative path name in place of a simple directory name.

REMOVING DIRECTORIES *(rmdir)*

If you decide you no longer need a directory, you can remove it with the **rmdir** command. The **rmdir** command, which stands for **rem**ove a **dir**ectory, removes a directory if that directory does not contain subdirectories and files, or, in other words, if the directory is empty.

If the directory you are attempting to remove is not empty, **rmdir** will not remove it unless you remove the contents of the directory first. In addition, you are not allowed to remove directories belonging to other system users unless you have permission to do so.

The standard format for the **rmdir** command is:

rmdir *directoryname(s)<CR>*

where one or more directory names can be specified.

If you were to attempt to remove the directory **bin** in the sample file system, the ROS system would respond in the following manner:

```
$ rmdir bin<CR>
rmdir: bin not empty
$
```

To remove the directory **bin** with the **rmdir** command, you would first have to remove the files **display** and **list** and the subdirectory **tools**. If you wish to remove files, see the section entitled *Accessing and Manipulating Files* in this chapter. To remove any subdirectories like **tools**, use the **rmdir** command. The system will return the $ prompt in response to the **rmdir** command when the directory specified in the command line is empty.

The command recap that follows summarizes how **rmdir** works.

Command Recap

rmdir - remove a directory

command	*options*	*arguments*
rmdir	none	**directoryname(s)**
Description:	**rmdir** removes named directories if they do not contain files and/or subdirectories.	
Remarks:	If the directory is empty, the system returns the $ prompt when the directory is removed. If the directory contains files or subdirectories, the message, **rmdir:** *directory name* **not empty** is returned to you.	

ACCESSING AND MANIPULATING FILES

This section introduces you to several ROS system commands that access and manipulate files in the file system structure. Information in this section is organized into two parts--basic and advanced. The part devoted to basic commands is fundamental to your using the file system; the advanced commands offer you more sophisticated information processing techniques when working with files. You may skip reading the advanced section if you do not need to use the commands it covers.

BASIC COMMANDS

This section discusses ROS system commands that are important to your being able to access and use the files in your directory structure. Specifically, these commands and their capabilities are:

cat -- Outputs the contents of a file you name

pg -- Prints on a video display terminal the contents of a file you name in chunks or pages

pr -- Prints on your terminal a partially formatted version of the file you name

lpr -- Allows you to produce a paper copy of a file on a line printer

cp -- Makes a duplicate copy of an existing file

mv -- Moves and renames a file

rm -- Permanently removes a file when you no longer need it

wc -- Counts the lines, words, and characters in a file

chmod --Changes permission modes for a file (and a directory)

Each command is covered in one of following sections. A command recap follows the discussion of each command allowing you to review quickly the command line syntax and command capabilities.

Displaying a File's Contents *(cat, pg, pr)*

The ROS system provides three commands that allow you to display and print the contents of a file or files--**cat, pg,** and **pr**. The **cat** command, which stands for con**cat**enate, outputs the contents of files you specify by name on the command line, and displays the result on your terminal unless you tell **cat** to direct the output to another file or a new command. The **pg** command is particularly useful when you wish to read the contents of a lengthy file or a number of files because the command displays

the text of a file in chunks or pages, a screenful at a time at your direction on a video display terminal. The **pr** command partially formats and outputs the files you specify on your terminal unless you direct the output to a paper printing device (see the **lpr** command in this chapter).

The following three sections describe how to use these commands.

Concatenate and Print Contents of a File *(cat).* The **cat** command displays the contents of a file or files. For example, if you are located in directory **letters** in the sample file system and you wish to display the contents of the file **johnson**, you would type **cat johnson<CR>** and the following output would appear on the terminal.

```
$ cat johnson<CR>
This file contains a letter
to Mr. Johnson on the topic of
office automation.
$
```

As you can see, the contents of the file are displayed after the command line and are followed by the $ prompt.

To display the contents of two (or more) files, like **johnson** and **sanders**, simply type **$ cat johnson sanders<CR>** and the **cat** command reads **johnson** and **sanders** and displays their contents in that order on your terminal.

```
$ cat johnson sanders<CR>
This file contains a letter
to Mr. Johnson on the topic of
office automation.
This file contains a letter
to Mrs. Sanders inviting her to
speak at our departmental
meeting.
$
```

To direct the output of the **cat** command to another file or to a new command, see the *REDIRECTING INPUT AND OUTPUT* section in Chapter 6.

The command recap that follows summarizes what you can expect the **cat** command to do.

Command Recap

cat - concatenate and print a file's contents

command	*options*	*arguments*
cat	available*	**filename(s)**
Description:	**cat** reads the name of each file given on the command line and displays the contents of the files.	
Remarks:	If the file(s) exist, the contents are displayed on the terminal monitor; if not, the message **cat: cannot open filename** is returned to you. If you wish to display the contents of a directory, use the **ls** command.	

* See the *ROS Reference Manual* for all available options and an explanation of their capabilities.

Paging Through the Contents of a File *(pg).* The **pg** command, short for **page**, allows you to examine the contents of a file or files screenful by screenful on a video display terminal. The **pg** command displays the text of a file in chunks or pages followed by a colon (:). After displaying the colon, the system pauses and waits for your instructions to proceed. For example, your instructions can request **pg** to continue displaying the file's contents a page at a time or you can ask **pg** to search through the file(s) to locate a specific character pattern. Table 4-1 summarizes some of the instructions you can give **pg** after the colon is displayed.

Table 4-1

Summary of Selected Commands for pg*	
Command†	**Meaning**
h	Help; display list of available **pg** commands
q *or* Q	Quit **pg** perusal mode
<CR>	Display next page of text
l	Display next line of text
d or ^d	Display additional half page of text
. or ^l	Redisplay current page of text
f	Skip next page of text, and display following one
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

* See the *ROS Reference Manual* for a detailed explanation of all available **pg** commands.

† Most commands can be typed with a number preceding them: +1 (display next page), −1 (display previous page), or 1 (display first page of text).

The **pg** command is especially useful when you wish to peruse a long file or a series of files because the system pauses after displaying each page allowing you as much time as you need to examine it. The size of the page displayed depends on the terminal you are using. For example, on a video display terminal with a window capable of showing 24 lines, 23 lines of text and a line containing the colon will be displayed as a page. However, if the file is less than 23 lines long, the page size will be the number of lines in the file plus the line containing the colon.

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

pg *filename(s)*<**CR**>

For example, to display the contents of the file **outline** in the sample file system, type **pg outline**<**CR**> and the first page of the file will appear on the screen. Since the file has more lines in it than can be displayed in one page, the colon indicates there is more to be looked at when you are ready. Pressing the <**CR**> key will print the next page of the file.

The following screen summarizes what has been done thus far.

```
$ pg outline<CR>
After you analyze the subject for your
report, you must consider organizing and
arranging the material you wish 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
:<CR>
```

After pressing the <CR> key, the **pg** program will resume outputting the file's contents on the screen as follows:

```
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):
```

In addition to the remainder of the file's contents, a line with the output **(EOF):** is displayed. The EOF designates that you have reached the end of the file and the colon is your cue for the next instruction.

When you have completed examining the file, you can type **q** or **Q** followed by pressing the <CR> key and the $ prompt will appear on your screen. Or you can choose to use one of the other available commands given in Table 4-1 depending on your needs.

In addition, there are a number of options that can be specified on the **pg** command line. Refer to the *ROS Reference Manual* if you are interested in learning more about them.

The following command recap summarizes the highlights of **pg**'s capabilities.

Command Recap

pg - display a file's contents in chunks or pages

command	*options*	*arguments*
pg	available*	**filename(s)**
Description:	**pg** reads the name of each file given on the command line and displays the contents of the file(s) in chunks or pages, screenful by screenful.	
Remarks:	After displaying a screenful of text, the **pg** command awaits your instruction to continue to display text, to search for a pattern of characters, or to exit the **pg** perusal mode. In addition, a number of options are available for you to use with **pg** on the command line. For example, you can start to display the contents of file at a specific line or at a line containing a certain sequence or pattern or you can opt to go back and review text that has already been displayed.	

* See the *ROS Reference Manual* for all available options and an explanation of their capabilities.

The **more** command is similar to the **pg** command, but offers additional capabilites. See the *more*(1) pages in the *ROS Reference Manual* for details.

Print Partially Formatted Contents of a File *(pr).* The **pr** command is typically used to prepare files for printing. You can expect the **pr** command to title, paginate, supply headings, and print a file according to varying page lengths and widths on your terminal monitor unless you specify that it prints on another output device, such as a line printer (read the discussion on the **lpr** command in this section), or you direct the printing to a different file (see the section *REDIRECTING INPUT AND OUTPUT* in Chapter 6).

If you choose not to specify any of the available options, the **pr** command produces output that is in a single column with 66 lines per page and is preceded by a short heading. The heading consists of five lines--two blank lines; a line containing the date, time, file name, and page number; and two more blank lines. And the formatted file is followed by five blank lines. (Complete sets of text formatting tools, called **nroff** and **troff**, are available for your ROS system.)

Typically, the **pr** command is used in tandem with the **lpr** command to provide a paper copy of text as it was entered into a file. (See the section discussing the **lpr** command for details.) However, you can also use the **pr** command to format partially and print the contents of a file on your terminal. For example, to review the contents of the file **johnson** in the sample file system, type in the command **pr johnson<CR>**. The following screen summarizes this activity.

```
$ pr johnson<CR>

Nov 29 09:19 1983  johnson Page1

This file contains a letter
to Mr. Johnson on the topic of
office automation.
.
.
.
$
```

Note that the ellipses after the last line in the file stand for the remaining 58 lines (all blanks in this case) that **pr** formatted into the output. If you are working on a video display terminal, which typically allows you to view about 24 lines at a time, the entire 66 lines of the formatted file will print continuously and rapidly to the end of file. This means that the first 41 lines will "roll" off the top of your screen making it impossible for you to read them unless you have the ability to "roll" or "page" back a screen or two. If you are looking at a particularly long file, this feature might not solve the problem.

In this case, you should use the control-s **<^s>** combination to stop printing on your terminal temporarily and control-q **<^q>** to resume the printing.

The command recap that follows summarizes what you can expect the **pr** command to do.

Command Recap

pr - print partially formatted contents of a file

command	*options*	*arguments*
pr	available*	**filename(s)**
Description:	**pr** produces a partially formatted copy of a file(s) on your terminal monitor unless otherwise specified. The program prints the text of the file(s) on 66-line pages and places five blank lines at the bottom of each page and a five-line heading at the top of each page. The heading consists of two blank lines; a line containing the date, time, file name and page number; and two additional blank lines.	
Remarks:	If the specified file(s) exists, the contents are partially formatted and displayed on the screen; if not, the message **pr: can't open filename** is returned to you. The **pr** command is most commonly used with the **lpr** command when a paper copy of a file is needed. However, when using the **pr** command to review a file on a video display terminal, use <^**s**> and <^**q**> to temporarily stop and start printing the file.	

* See the *ROS Reference Manual* for all available options and an explanation of their capabilities.

Printing a File *(lpr)*

At some point in time, you may want a paper copy of a file. In this case, if you have a printer connected to your Ridge, simply turn the printer on and use **cat** or **pr** to print the file. If, however, you wish to obtain a higher quality paper copy, you will want to use the **lpr** command. The **lpr** command, which stands for **l**ine **pr**inter, allows you to request your printer to furnish you with a paper copy of a file or files (Figure 4-7).

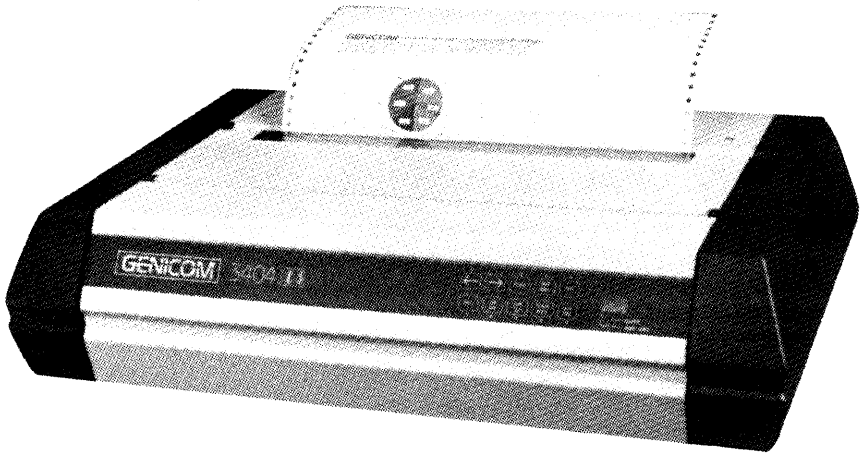


Figure 4-7. Line Printer

The line printer or types of line printers that you have access to depends on your specific Ridge 32 installation. You should ask your system administrator for the names of the printers available to you.

The basic format for the command is:

lpr *file*<**CR**>

For example, to print the file *letters* on a line printer, you would type **lpr letters**<**CR**> on the command line. In turn, the system would provide you with the name of the device or type of device on which the file will be printed and an identification (id) number indicating your request. The following screen summarizes this activity.

```
$ lpr letters<CR>
Request id is 6885   1 file
$
```

The system response indicates that your job is to be printed on a line-printing device (the system default), has a request id number of 6885, and is to include the printing of one file.

Using the **-m** option would cause mail to be sent to you indicating when the job is completed.

If you would like to cancel the request to **lpr** to print the file *letters*, use the format:

lprm *file*<CR>,

where *file* is the request id, the filename or the owner.

The **lpq** command gives the status and request id of the line printer jobs.

A command recap follows that summarizes what you can expect of the **lpr** command.

Command Recap

lpr - request paper copy of file from a line printer

command	*options*	*arguments*
lpr	**–m,** and others*	**file(s)**
Description:	**lpr** requests that specified files be printed by a line printer, thus providing paper copies of the contents.	
Options:	**–m** Sends a message to you via **mail** after the printing is complete.	
Remarks:	You can cancel a request to the line printer by typing **cancel** and the request **id** furnished to you by the system when the request was acknowledged.	
	Check with the system administrator for information on additional and/or different commands for printers that may be available at your location.	

* See the *lpr*(1) pages in the *ROS Reference Manual* for all available options and an explanation of their capabilities.

The **lpr** command is also available with the optional lp spooling system. See the *lpr*(1) pages in the *ROS Reference Manual* for details.

Making a Duplicate Copy of a File *(cp)*

When using the ROS system, you may wish to make a copy of a file. For example, you might want to revise a file while leaving the original version intact. The ROS system provides you with the **cp** command, short for **copy**, which copies the complete contents of one file into another. The **cp** command also allows you to copy one or more files from one directory into a different directory while leaving the original file or files in place.

To copy the file named **outline** to a file named **new.outline** in the sample directory, simply type **cp outline new.outline<CR>**. The system returns the $ prompt when the copy is made. To verify the existence of the new file, you can type **ls<CR>**, which lists the names of all files and directories in the current directory, in this case **draft**. The following screen summarizes the activity.

```
$ cp outline new.outline<CR>
$ ls<CR>
new.outline   outline   table
$
```

You know from looking at the sample file system that the file **new.outline** did not exist before the **cp** command to copy **outline** to **new.outline** was given. However, if it had, it would have been replaced by a copy of the file **outline** and the previous version of **new.outline** would have been deleted.

If you had tried to copy the file **outline** to another file named **outline** in the same directory, the system would have told you that the file names were identical and returned the $ prompt to you. If you listed the contents of the directory to determine exactly how many copies of **outline** exist, the terminal monitor would look something like the following:

```
$ cp outline outline<CR>
cp: outline and outline are identical
$ ls<CR>
outline   table
$
```

As you can see, the ROS system does not allow you to have two files with the same name in a directory.

You could, however, copy the file named **outline** from the directory **draft** to another file named **outline** in the directory named **letters** by using any of the following command lines assuming you are currently in **draft**:

```
cp outline ../letters/outline<CR>
cp outline ../letters<CR>
cp outline /user1/starship/letters/outline<CR>
cp outline /user1/starship/letters<CR>
```

A copy of the file **outline** would reside in both directories **draft** and **letters** after using one of these commands since each of them contains a legal path name to the file **outline**. From this example, you can see that the ROS system allows you to have files with identical names as long as they are in different directories.

If you would like to copy the file **outline** in the directory **draft** to a file named **outline.vers2** in the directory **letters**, you could use either of the following command lines:

```
cp outline ../letters/outline.vers2<CR>
cp outline /user1/starship/letters/outline.vers2<CR>
```

The following recap summarizes how the **cp** command works.

Command Recap

cp - make a copy of a file

command	*options*	*arguments*
cp	none	**file1 file2** **file(s) directory**
Description:	**cp** allows you to make a copy of **file1** and call it **file2** leaving **file1** intact, or to copy one or more files into a different directory.	
Remarks:	When copying **file1** to **file2** and **file2** already exists, the **cp** command will overwrite the first version of **file2** with a copy of **file1** calling it **file2**. The first version of **file2** is deleted. You cannot copy directories with the **cp** command.	

Moving and Renaming a File *(mv)*

The **mv** 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 in a directory, use the following command:

mv *file1 file2*<**CR**>

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

For example, if you are in the directory **draft** in the sample file system and you would like to rename the file **table** as **new.table**, simply type **mv table new.table<CR>**. You should receive the $ command prompt if the command executed successfully. To verify that the file **new.table** exists, you can list the contents of the directory by typing **ls<CR>**. In turn, the terminal should read:

```
$ mv table new.table<CR>
$ ls<CR>
new.table    outline
$
```

You can also move a file from one directory to another keeping the file's name the same or changing it to a different one. To do so, use the following command line.

mv *file(s) directory*<**CR**>

where the file and directory names can be any valid names, including path names.

To move the file **table** from the current directory named **draft** (whose full path name is **/user1/starship/draft**) to a file with the same name in the directory **letters** (whose relative path name from **draft** is **../letters** and whose full path name is **/user1/starship/letters**), any one of several command lines can be used, including the following:

mv table /user1/starship/letters<CR>
mv table /user1/starship/letters/table<CR>
mv table ../letters<CR>
mv table ../letters/table<CR>
mv /user1/starship/draft/table /user1/starship/letters/table<CR>

The file **table** could have been renamed **table2** when moving it to the directory **letters** using any of the following:

mv table /user1/starship/letters/table2<CR>
mv table ../letters/table2<CR>
mv /user1/starship/draft/table2 /user1/starship/letters/table2<CR>

You can verify that the command worked by listing the contents of the directory with the **ls** command.

Refer to the recap that follows for a summary of how the **mv** command works.

Command Recap

mv - move or rename files

command	*options*	*arguments*
mv	none	**file1 file2** **file(s) directory**
Description:	**mv** allows you to change the name of a file or to move a file(s) into another directory.	
Remarks:	When changing the name of **file1** to **file2** and **file2** already exists, the **mv** command will overwrite the first version of **file2** with **file1** and rename it **file2**. The first version of **file2** is deleted.	

Removing a File *(rm)*

When you no longer need a file, you can get rid of it by using the **rm** command, which is short for **rem**ove.

To remove one or more files, use the format:

rm *file(s)*<**CR**>

After the command executes, the file(s) you specified are removed permanently.

To remove a file named **new.outline** in the current directory type **rm new.outline<CR>** and list the contents of the directory with the **ls** command to verify that the file **new.outline** no longer exists.

To remove more than one file, such as the files **outline** and **table**, type **rm outline table<CR>** and list the contents of the directory by typing **ls<CR>**.

```
$ rm outline table<CR>
$ ls<CR>
$
```

The $ response indicates that the files named **outline** and **table** were removed permanently.

The following recap summarizes how the **rm** command works.

Command Recap

rm - remove a file

command	*options*	*arguments*
rm	available*	**file(s)**
Description:	**rm** allows you to remove one or more files.	
Remarks:	Files specified as arguments to the **rm** command are removed permanently.	

* See the *ROS Reference Manual* for all available options and an explanation of their capabilities.

Counting Lines, Words, and Characters in a File *(wc)*

The **wc** command, which stands for **w**ord **c**ount, reports the number of lines, words, and characters there are in a file that you specify by name 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. In addition, you can direct the **wc** program to give you only a line, a word, or a character count by using the **–l**, **–w**, or **–c** options, respectively.

To determine the number of lines, words, and characters in a file, use the following format on the command line:

wc *file1*<CR>

When you do, the system responds with a line in the format:

l w c file1

where

l = Number of lines in *file1*,

w = Number of words in *file1*, and

c = Number of characters in *file1*.

For example, to count the lines, words, and characters in the file **johnson** in the current directory **letters**, type **wc johnson<CR>**. The terminal monitor would show the following output:

```
$ wc johnson<CR>
      3     14     78   johnson
$
```

The system response displays the line count (**3**), the word count (**14**), and the character count (**78**) for the file **johnson**.

To determine the number of lines, words, and characters in more than one file, use the following format:

wc *file1 file2*<**CR**>

In turn, the system responds with the following format:

l w c file1
l w c file2
l w c total

where line, word, and character counts are displayed for **file1** and **file2** on separate lines and the combined counts appear on the last line called **total**.

If you request that the **wc** program count lines, words, and characters in the files **johnson** and **sanders** in the current directory, the system would respond as follows:

```
$ wc johnson sanders<CR>
      3     14     78   johnson
      4     16     95   sanders
      7     30    173   total
$
```

In this case, the first line of the system response shows the line, word, and character counts for the file **johnson**. The second line of output gives line, word, and character

counts for **sanders**. The last line of output shows combined line, word, and character counts for both files in the line labeled **total**.

If you prefer to get only a line, a word, or a character count, select the appropriate format from the following lines:

wc	**–l**	**file1<CR>**	(line count)
wc	**–w**	**file1<CR>**	(word count)
wc	**–c**	**file1<CR>**	(character count)

For instance, by typing **wc –l sanders<CR>** on the command line you would obtain the following output:

```
$ wc –l sanders<CR>
      4  sanders
$
```

The system tells you that the number of lines in the file **sanders** is 4 in answer to specifying **–l**. If the **–w** or **–c** option was specified for that file, the ROS system would have responded with the number of words or number of characters, respectively, in the file.

The command recap that follows summarizes how the **wc** command works.

Command Recap

wc - count lines, words, and characters in a file

command	*options*	*arguments*
wc	**–l, –w, –c**	**file(s)**
Description:	**wc** counts lines, words, and characters in the file(s) named keeping a total count of all tallies when more than one file is specified.	
Options	**–l** Counts the number of lines in the specified file(s).	
	–w Counts the number of words in the specified file(s).	
	–c Counts the number of characters in a specified file(s).	
Remarks:	When a file name is specified in the command line, it is printed with the count(s) requested.	

Protecting Your Files *(chmod)*

The **chmod** command, short for **ch**ange **mod**e, allows you to decide who can read, alter, and use your files and who cannot. Because the ROS operating system is a multiuser system, you are not working alone in the file system: you and other system users can follow path names and run system commands to move to various directories and to read and use files belonging to one another if you have permission to do so.

If you own a file, then you are able to determine who has the right to read that file, to make changes to or write the file, and to run or execute the file if it is a program. These permissions are defined as:

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

w = Allows system users to write changes into a file or copy of a file, and

x = Permits system users to run an executable file.

Specifically, you can determine who in the community of ROS system users is entitled to these various permissions and who is not according to the following classifications:

u = You, the user and login owner of your files and directories,

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 set up your ROS system account), and

o = All other system users.

When you create a file or a directory, the system automatically grants or denies permission specifically to you, members of your group, and other system users. You can alter this automatic action to some extent by modifying your environment, which is discussed in Chapter 6. Regardless of how the permissions are granted when a file is created, as the owner of the file or directory it is up to you to allow current permissions to remain in effect or to change them to suit your purposes and the situation. For example, you may wish to keep certain files private and for your use only. Or you may wish to grant permission to read and to 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 command that produces a long listing of a directory's content, which is **ls –l**. For example, typing **ls –l<CR>** while in the directory named **starship/bin** in the sample file system would produce the following output:

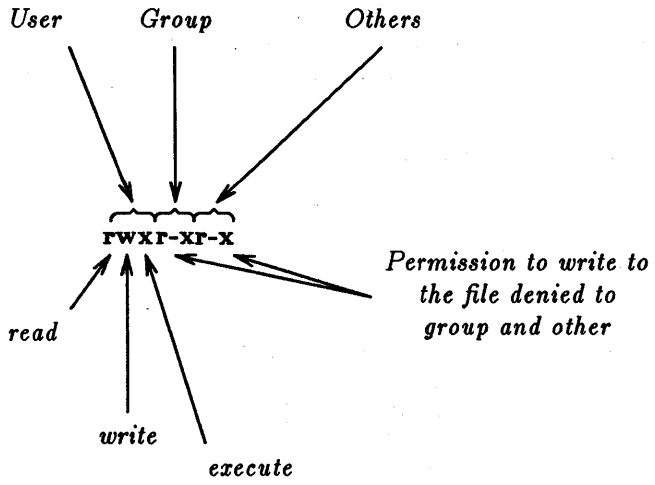
```
$ ls -l<CR>
total 35
-rwxr-xr-x  1   starship project  9346    Nov 1  08:06  display
-rwx--x--x  1   starship project  6428    Dec 2  10:24  list
drwx--x--x  2   starship project    32    Nov 8  15:32  tools
$
```

Permissions for the files **display** and **list** and the directory **tools** are shown on the left of the terminal monitor under the line **total 35**, and look like:

```
rwxr-xr-x     (file display)
rwx--x--x     (file list)
rwx--x--x     (directory tools)
```

These nine characters represent three groups of three characters. The first set of three characters refers to your (or the user's/owner's) permissions, the second set to members of the group, the last set to all other system users. Within each set of characters, the **r**, **w**, and **x** indicate the permission currently enabled for the groups. If a dash appears instead of an **r**, **w**, or **x**, permission to read, write, or execute is denied.

The following diagram summarizes this breakdown for the file named **display**.



As you can see, the owner has **r**, **w**, and **x** permissions and members of the group and other system users have **r** and **x** permissions.

How to Change Existing Permissions. After you have determined what permissions are in effect, you can change them using the following format:

chmod *who* + **(or** −**)** *permission file(s)*<**CR**>

where:

chmod = Name of program,

who = One of three user groups **u**, **g**, **o**:
u = User,
g = Group, and
o = Other.

+ − = Instruction that grants (+) or denies (−) permission.

permission = Authorization to **r**, **w**, or **x**:
r = Read,
w = Write, and
x = Execute.

file(s) = File (or directory) name(s) listed; assumed to be branches from your working directory, unless you use full path (names).

This may sound a bit confusing. But, a few examples on how to use the **chmod** command should help to make permission possibilities clear.

Let's use the permissions for the file **display** to experiment with **chmod**. You can see from the permissions that as the user and owner of **display** you can read, write, and run this executable file. You can protect the file against accidentally changing it by denying yourself write (**w**) permission by typing the command line **chmod u−w display**<**CR**>. After receiving the $ prompt, type in **ls −l**<**CR**> to verify the permission has changed.

```
$ chmod u-w display<CR>
$ ls -l<CR>
total 35
-r-xr-xr-x  1   starship project  9346    Nov 1  08:06  display
-rwx--x--x  1   starship project  6428    Dec 2  10:24  list
drwx--x--x  2   starship project    32    Nov 8  15:32  tools
$
```

From this output, you can see that you no longer have permission to write changes into the file, that is, unless you change the mode back to include write permission.

Now, let's consider another example. Notice that permission to write into the file **display** has been denied to members of your group and other system users. These users, however, have read permission, which means that any of these users can copy the file into their own directories and then make changes to it. To prevent all system users from copying this file, you could deny them read permission by typing **chmod go–r display<CR>**. 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.

```
$ chmod go–r display<CR>
$ ls –l<CR>
total 35
-rwx--x--x  1   starship project  9346    Nov 1  08:06  display
-rwx--x--x  1   starship project  6428    Dec 2  10:24  list
drwx--x--x  2   starship project    32    Nov 8  15:32  tools
$
```

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

The impact, however, of granting or denying permissions for directories to various system users is worth considering. For example, if 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 can read the names of the files that directory contains by using the **ls –l** command. Similarly, granting write permission allows the designated users to create new files in the directory and change and remove existing ones. And granting permission to execute the directory allows the designated users the ability to move to that directory (and make it their working directory) by using the **cd** command.

An Alternate Method. The **chmod** method described in the preceding pages is one of two ways to change permissions to read, write, and execute files and directories. The method previously described uses symbols, such as *r*, *w*, *x* and *u*, *g*, *o*, to specify instructions to **chmod**. Hence, it is called the symbolic method.

The alternate method uses a number system called octal that is different than the decimal number system we typically use on a day-to-day basis. This method uses three octal numbers ranging from 0 through 7 to assign permissions. If you wish to use the octal method when changing permission, see the description of **chmod** in the *ROS Reference Manual.*

Summary. The command recap that follows provides a quick reference on how **chmod** works.

Command Recap

chmod - change permission modes for files (and directories)

command	*instruction*	*arguments*
chmod	**who + − permission**	**filename(s) directoryname(s)**
Description:	**chmod** gives (+) or removes (−) *read, write,* and *execute* permissions for three types of system users: **user** (you), **group** (members of your group), and **other** (all other users able to access the system on which you are working).	
Remarks:	The instruction set can be represented in either octal or symbolic terms.	

ADVANCED COMMANDS

You will become more and more familiar with the file system as you use the commands thus far discussed in this chapter. As this familiarity increases so might your need or interest for more sophisticated information processing techniques when working with files. This section introduces you to three commands that give you just that. These commands and their capabilities are listed as follows:

diff -- Finds difference between two files,

grep -- Searches a file for a pattern, and

sort -- Sorts and merges files.

The following discussion only scratches the surface on information processing techniques available with the ROS system. You may refer to the *ROS Reference Manual* for additional information.

Identifying Differences Between Files *(diff)*

The **diff** command locates all the differences between two files and proceeds to tell you how to change the first file to be a carbon copy of the second. It reports all differences between the files.

The basic format for the command is:

diff *file1 file2*<CR>

If **file1** and **file2** are identical, the system returns the $ prompt to you. If not, the **diff** command instructs you on how to bring the first file into agreement with the second by using line editor (**ed**) commands. (See the **ROS Text Editing Guide** for details on the line editor.) The ROS system flags lines in **file1** with the < symbol and **file2** with the > symbol.

For example, if you use the **diff** command to identify differences between the files **johnson** and **sanders**, the system would respond as follows:

```
$ diff johnson sanders<CR>
2,3c2,4
< to Mr. Johnson on the topic of
< office automation.
---
> to Mrs. Sanders inviting her to
> speak at your departmental
> meeting.
$
```

The first line of the system response is

2,3c2,4

which means lines 2 through 3 in the file **johnson** must be changed (designated by **c**) to lines 2 through 4 in the file *sanders*.

The system then displays lines 2 through 3 in the file **johnson** as follows:

```
< to Mr. Johnson on the topic of
< office automation.
```

and lines 2 through 4 in the file **sanders**

```
> to Mrs. Sanders inviting her to
> speak at our departmental
> meeting.
```

If you make these changes (using the **ed** or the **vi** text editing program), the file **johnson** will be identical to the file **sanders**. Remember, the **diff** command tells you exactly what the differences are between the named files. If you simply want an identical copy of a file, use the **cp** command.

Refer to the recap that follows for a summary of what you can expect the **diff** command to do when no options are specified. See the reference to the *ROS Reference Manual* for details on available options.

Command Recap

diff - finds differences between two files

command	*options*	*arguments*
diff	available*	**file1 file2**
Description:	**diff** reports what lines are different in two files and what you must do to make the first file identical with the second.	
Remarks:	Instructions on how to change a file to bring it into agreement with another file are line editor (**ed**) commands: **a** (append), **c** (change), or **d** (delete). Numbers given with **a**, **c**, or **d** indicate the lines to be modified. Also used are the symbols < (indicating a line from the first file) and > (indicating a line from the second file).	

* See the *ROS Reference Manual* for all available options and an explanation of their capabilities.

Searching a File for a Pattern *(grep)*

You can request the ROS system to search through files for a specific word, phrase, or group of characters by using the **grep** command. Technically, **grep** means **g**lobally search through a file or files to locate a **r**egular **e**xpression and **p**rint the lines that contain the regular expression. Put simply, a regular expression is the pattern of characters--be it a word, a phrase, or an equation--that you specify.

The basic format for the command line is:

grep *pattern file(s)*<CR>

Thus, to locate the line containing the word **automation** in the file **johnson**, you would type:

grep automation johnson<CR>

and the system would respond as follows:

```
$ grep automation johnson<CR>
office automation
$
```

The output gives you all the lines in the file **johnson** that contain the pattern for which you were searching, which is the word **automation**.

If the pattern contains multiple words or any characters that have a special meaning to the ROS system, such as $, |, *, ?, and so on, the entire pattern must be enclosed in single quotes. (For an explanation of the special meaning for these and other characters see the section entitled **Metacharacters** in Chapter 6.) For example, if you are interested in locating the lines containing the pattern **office automation**, the command line and system response would read:

```
$ grep 'office automation' johnson<CR>
office automation.
$
```

But what if you could not recall to whom you sent a letter on the topic of office automation in the first place--Mr. Johnson or Mrs. Sanders? You could type:

grep 'office automation' johnson sanders<CR>

If you did, the system would respond in the following manner:

```
$ grep 'office automation' johnson sanders<CR>
johnson:office automation.
$
```

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

In addition to the capabilities of the **grep** command that are summarized in the recap that follows, the ROS system provides variations to the basic **grep** command, called **egrep** and **fgrep**, along with several options that further enhance the searching powers of the command. See the *ROS Reference Manual* if you are interested in learning more.

Command Recap

grep - searches a file for a pattern

command	*options*	*arguments*
grep	available*	**pattern file(s)**
Description:	**grep** searches the file or files you name for lines containing a pattern and then prints the lines that match. If you name more than one file, the name of the file containing the pattern is given also.	
Remarks:	If the pattern you give contains multiple words or special characters, enclose the pattern in single quotes on the command line.	

* See the *ROS Reference Manual* for all available options and an explanation of their capabilities.

Sorting and Merging Files *(sort)*

The ROS system provides you with an efficient tool called **sort** for sorting and merging files. The basic form of the command line is:

sort *file(s)*<**CR**>

which causes lines in the specified files to be sorted and merged in the order defined by the ASCII representations of the characters in the lines.

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

To get an idea of how the **sort** command works, let's say that you have two files, named **phase1** and **phase2**, each containing a list of names that you wish to sort alphabetically

and finally interfile into one list. First, display the contents of each file using the **cat** command.

```
$ cat phase1<CR>
Smith, Allyn
Jones, Barbara
Cook, Karen
Moore, Peter
Wolf, Robert
$ cat phase2<CR>
Frank, M. Jay
Nelson, James
West, Donna
Hill, Charles
Morgan, Kristine
$
```

(Note: we could have used the command line **cat phase1 phase2<CR>** instead of listing the contents of each file separately.)

Now, sort and merge the contents of the two files using the **sort** command. Note that the output of the **sort** program will print on the terminal monitor unless you specify otherwise.

```
$ sort phase1 phase2<CR>
Cook, Karen
Frank, M. Jay
Hill, Charles
Jones, Barbara
Moore, Peter
Morgan, Kristine
Nelson, James
Smith, Allyn
West, Donna
Wolf, Robert
$
```

In addition to putting together simple listings as in the previous examples, the **sort** command can rearrange the lines and parts of lines (called fields) according to a number of other specifications you can designate on the command line. The possible specifications are complex and are not within the scope of this text. You should consult the *ROS Reference Manual* for a full rundown on the available options.

However, the following command recap summarizes the capabilities of the **sort** program.

Command Recap

sort - sorts and merges files

command	*options*	*arguments*
sort	available*	**file(s)**
Description:	**sort** sorts and merges lines from the file or files you name and displays the result on your terminal.	
Remarks:	If no options are specified on the command line, lines are sorted and merged in the order defined by the ASCII representations of the characters in the lines.	

* See the *ROS Reference Manual* for all available options and an explanation of their capabilities.

SUMMARY

This chapter described the structure of the file system and presented ways to use and to navigate through the file system via ROS system commands. The next chapter gives you an overview of a variety of ROS system capabilities, such as text editing, using the shell as a command language, communicating electronically with other system users, and programming and developing software.

Chapter 5: ROS SYSTEM CAPABILITIES

Chapter 5
ROS SYSTEM CAPABILITIES

INTRODUCTION

The material in this chapter combines basic, fundamental concepts about the ROS system covered in Chapters 2, 3, and 4 of this guide with information about system capabilities that you may use to do your computing work efficiently and effectively.

This chapter provides an overview of the following ROS system capabilities: text editing, working in the shell, communicating electronically, and programming in the ROS system environment.

TEXT EDITING

You have read a good deal about files up to this point simply because using the file system is a way of life in a ROS system environment. The information in this section will enhance your knowledge about manipulating files by introducing you to a software tool called a text editor. A text editor provides you with the ability to create and modify files: it will help you to fare well in the ROS system since a considerable amount of your computing time may be spent writing and revising letters, memos, reports, or source code for programs that will be stored in files.

This section contains information that tells you what a text editor is and how it works. In addition, this section acquaints you with two types of text editors supported on the ROS system: the line editor and the visual, or screen, editor. Since you will probably come to prefer one of these editing programs over the other--even if you learn to use them equally well--the line editor and the screen editor are briefly compared to help you to assess their capabilities. For detailed information on the line editor and the screen editor, see the *ROS Text Editing Guide.*

THE TEXT EDITOR

When you write or type letters, memos, and reports and then decide to change what you have written or typed, you will use skills required in text editing. These skills include inserting new or additional material, deleting unneeded material, transposing material (sometimes called cutting and pasting), and finally preparing a clean, corrected copy. Text editors perform these tasks at your direction making writing and revising text much easier and quicker than if done by hand or on a typewriter.

In the ROS system, a text editor is much like the ROS system shell. Both a text editor and the shell are programs that accept your commands and then perform the requested functions--essentially, they are both interactive programs. A major difference between a text editor and the shell, however, is the set of commands that each recognizes. All the commands you have learned up to this point belong to the shell's command set. A text editor, on the other hand, has its own distinct set of commands that allow you to create, move, add, and delete text in files, as well as acquire text from other files.

TEXT EDITOR OPERATION

To understand how a text editor works you need information about the environment created when you use an editing program and the modes of operation understood by a text editor.

Text Editing Buffers

To create a new file, you must ask the shell to put the editor in control of your computing session. When you do, a temporary work space is allocated to you by the editor. This work space is called the editing buffer; in it you can enter information you want the file to hold and modify it if you wish.

Because you are in a temporary work space when using a text editor, the file you are creating along with the changes you make to it are also temporary. This work space allotment and what it is holding will exist only as long as you work in the editing program. If you wish to save the file, you must tell the text editor to write the contents of the buffer into a storage area. If you do not tell the editor to write or record what you have done during the editing session, the buffer's contents will disappear when you leave the editing program. If you forget to write a new file or update an existing one, the text editors remind you to do so when you attempt to leave the editing environment.

To modify an existing file, the procedure is almost identical to the one you follow when creating a new file. First, call the editor and give it the name of the file you wish to change. In turn, the editor makes a copy of the file that is in the storage area and places it in the buffer so you can work on it.

When you finish editing the file, you can write the buffer's contents into storage and leave the editing program knowing the file is updated and ready to be recalled when you need it again. Or you can chose to leave the editor without writing the file if you have made a critical mistake or you are unhappy with the edited version. This step leaves the original file intact and the edited copy disappears.

Regardless of whether you are creating a new file or updating an existing one, the text you put in the buffer is organized into lines. A line of text is simply the series of characters that appears horizontally across a row of typing that is ended by pressing the <**CR**> key. Occasionally, files may contain a line of text that is too long to fit on the terminal monitor. Some terminals will automatically display the continuation of the line on the next row of the monitor, whereas others will not.

Modes of Operation

Text editors are capable of understanding two modes of operation: the command mode and the text input mode.

When you begin an editing session, you will automatically be placed in command mode. In command mode, all your input is interpreted as a command. Typical editing commands allow you to move about in a file, search for patterns in the file's contents, or print a portion of a file on the terminal monitor. The input mode is entered when you use a command to create text. Once in input mode, what you type on the keyboard is placed into the buffer as part of the text file until you send the appropriate instruction to the editor that returns you to command mode.

LINE EDITOR

The line editor, accessed by the **ed** command, is a fast, versatile program for preparing text files. This editor gets its name because it operates on the lines of text a file holds. For example, to change a single character in a file, you specify the line of the file that contains the character you wish to change and then specify the change.

Put simply, you manipulate text on a line-by-line basis with the line editor. Commands for this text editor can change lines, print lines, read and write files, and initiate text entry. In addition, you can specify the line editor to run from a shell program; something you cannot do with the screen editor. (See Chapter 6 for information on basic shell programming techniques.)

The line editor works equally well on paper printing terminals and video display terminals. It will also obligingly accommodate you if you are using a slow-speed telephone line.

Refer to the **ED** Section in the **ROS Text Editing Guide** for instructions on how to use this editing tool. If you are interested in a comparison of line editor (**ed**) and screen editor (**vi**) features, see Table 5-1.

Table 5-1
Comparison of Line (ed) and Screen (vi) Editors

Feature	Line Editor (ed)	Screen Editor (vi)
Recommended terminal type	Paper-printing or VDT*	VDT
Speed	Accommodates high- and low-speed data transmission lines.	Works best via high-speed data transmission lines (1,200+ baud).
Versatility	Can be specified to run from shell scripts as well as used during editing sessions.	Must be used interactively during editing sessions.
Sophistication	Changes text quickly. Uses comparatively small amounts of processing time.	Changes text easily. However, can make heavy demands on computer resources.
Power	Provides a reduced set of editing commands.	Provides its own editing commands and recognizes all line editor commands as well.

* VDT = video display terminal

SCREEN EDITOR

The screen editor, accessed by the **vi** command, is a display-oriented, interactive software tool. When you use the screen editor, your terminal acts as a window to let you view the file you are editing a screenful or page at a time. This editor works most efficiently and effectively when used on a video display terminal operating at 1,200 or higher baud.

For the most part, modifications to a file (such as, additions, deletions, and changes) are accomplished by positioning the cursor at the point in the window where the modification is to be made and then making the change. In other words, the screen editor displays the effects of editing changes in the context in which you make them. Because of this feature, the screen editor in considered to be much more sophisticated than the line editor.

Furthermore, the screen editor offers a replete collection of commands. For example, a number of screen editor commands allow you to move the cursor around within the window to a file. Other commands move the window up or down through a page or more of the file. Still other commands allow you to change existing text or to create new text. In addition to its own set of commands, the screen editor has access to all the commands offered by the line editor. This arsenal of commands accounts for the screen editor's tremendous power.

There is, however, a trade-off for the screen editor's speed, visual appeal, efficiency, and power, which is the heavy demand that it places on the computer's processing time. For example, a simple change might cause an entire screen to need updating. Moreover, if simple changes lead to long delays while you wait for a screen to be updated, the pleasant experience of using a visual-oriented editor can be somewhat diminished.

Refer to the *VI* section in the *ROS Text Editing Guide* for instructions on how to use this software. If you wish to compare the features of the line editor (**ed**) and the screen editor (**vi**), see Table 5-1.

WORKING IN THE SHELL

Every time you log into the ROS system you will be communicating directly with a program called the shell. You will continue to interact with the shell until you log off the system, unless you use a program, such as a text editor, that temporarily suspends your dealings with the shell until you are finished using that particular program.

The shell is much like other programs, except that instead of performing one job, as **cat** or **ls** does, it is central to most of your interactions with the ROS system. This is because the shell's primary function is to act as an interpreter between you and the Ridge 32 computer. As an interpreter, the shell translates your requests into language the computer understands, calls requested programs into memory, and executes them.

This section acquaints you with some of the ways you can use the Bourne shell (which is the default shell on your system) as the command language interpreter to simplify a computing session and to enhance your ability to use system features. In addition to running a single program for you, you can also use the shell to:

- interpret the name of a file or a directory you input in an abbreviated way using a type of "shell shorthand"
- redirect the flow of input and output of the programs you run
- execute multiple programs
- tailor your computing environment to meet your individual needs and preferences

In addition to being the command language interpreter, the shell is also a programming language. If you would like an overview of Bourne shell's programming capabilities, see the section entitled *Programming in the System* at the end of this chapter. Or refer to the Shell Tutorial in Chapter 6 for detailed information on how to use the shell as a command language interpreter and as a programming language. The *ROS Programmer's Guide*, should be consulted for complete, unabridged information on both Bourne shell and C-shell programming.

SHELL SHORTHAND

Many ROS system commands require that you name a file or a directory as an argument to it on a command line, such as **mkdir directory name(s)<CR>** or **rm filename(s)<CR>**. Easy enough! But suppose you have 12 files to remove corresponding to monthly reports for 1983 named **rept1**, **rept2**, **rept3**, **rept4**, and so on? Or suppose you need to move 24 files corresponding to file names **sect1**, **sect2**, ... **sect24** to a different directory?

Typing the file name for each monthly report after the **rm** command or the file name for each section after the **mv** command is still easy, but all the repetition gets tedious after entering four or five names.

In instances like these, you should consider using shorthand notation when specifying file or directory names. If the file or directory names have some part in common, you can use a type of shorthand to tell the shell that you are referring to all of them on the basis of the similarity without specifying each one individually. Or, if a file has a unique character or sequence of characters within a group of similarly named files, you can use this shorthand notation to locate the file on the basis of the difference.

The ROS system recognizes several characters as having special meanings when they are used in place of a directory name or when they appear as part of a file or directory name on a command line. These characters allow you to specify the names of files and directories in a rapid, abbreviated way. Some of the characters are referred to as metacharacters because of their special meanings to the shell.

The special characters are . .. ? * [ ] – \ and their meanings are summarized in Table 5-2. When you specify file or directory names, you can substitute various characters within them with the appropriate shorthand abbreviation. Any part of the name that is not a special character is taken at its literal value.

Table 5-2
Shorthand Notation for File and Directory Names

Special Character	Meaning	Reference
.	Current directory	*Chapter 4*
..	Parent directory	*Chapter 4*
?	Match any single character	*Chapter 6*
*	Match any number of characters	*Chapter 6*
[ ]	Designate a sequence of characters	
–	Specify a character range within [ ], such as A-Z	*Chapter 6*
\	Remove meaning of special characters	*Chapters 3, 6*

For example, for the possibilities described at the beginning of this section, typing **rm rept*<CR>** would remove all the files in the current directory starting with the characters **rept** followed by any other characters corresponding to monthly reports for 1983, and typing **mv sect* ../chapter3<CR>** would move all the files from the current directory beginning with the letters **sect** and followed by any other characters to another directory named **chapter3** belonging to its parent directory.

Details on how to use the special characters appear in other chapters of this guide as indicated in Table 5-2. Refer to that chapter for the information you need.

REDIRECTING THE FLOW OF INPUT AND OUTPUT

Up to this point in this guide, any request to ask the shell to execute a command was done by entering the command and the necessary argument(s) on the terminal keyboard. In turn, the output, if any, was displayed on the terminal monitor. This pattern illustrates the idea of standard input (stdin) and standard output (stdout).

In general, the place from which a program expects to receive its input is called the standard input. A ROS system command called **mail**, which you will learn more about in the *ROS Utility Guide*, provides a good example of this and warrants mentioning here. For example, to use **mail**, you would simply type **mail jmrs<CR>** and the **mail** command takes everything you type on your keyboard after **<CR>** until you type **<^d>** as input. After you type **<^d>**, mail sends your input to the person with the login name **jmrs**. The place to which a program writes its results, in this case the login name **jmrs**, is referred to as the standard output.

In the ROS system, most commands expect to receive their input from the keyboard and then display output on the terminal monitor. By default, then, the standard input is the keyboard and the standard output is the terminal monitor (Figure 5-1).

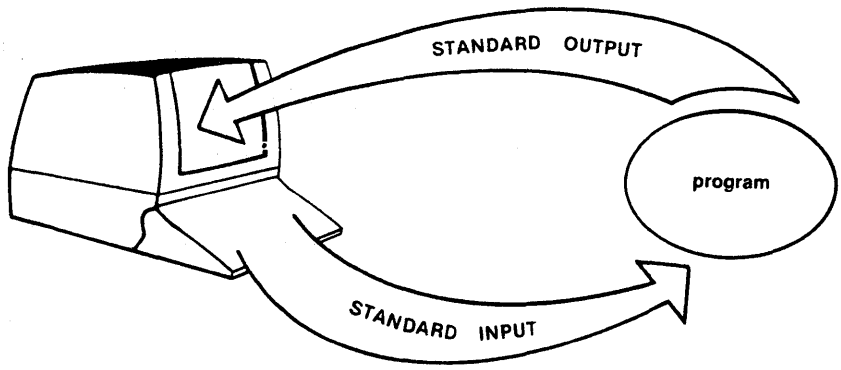


Figure 5-1. Standard Input/Output Flow

You can, if you wish, use a feature called redirection to change these defaults. Put simply, redirection is a ROS system feature that allows you to request the shell to reassign standard input and/or standard output to other files or devices.

With the redirection feature, you can request the shell to do the following:

- reassign to a file any output that a program would ordinarily send to your terminal
- have a program take its input from a file rather than from your terminal keyboard
- use a program as the source of data for another program

You request the shell to redirect input and output using a set of operators, which are > (greater than sign), >> (two greater than signs), < (less than sign), and | (a pipe). Now let's take a look at what each of these operators can do for you.

Redirecting the Standard Output *(>)*

The > operator allows you to redirect the output of a command (or program) into a file (Figure 5-2).

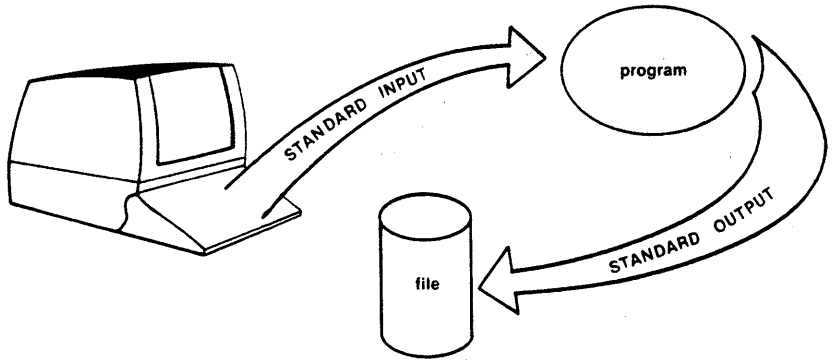


Figure 5-2. Redirecting Standard Output to File

To use the > operator, follow the command line format:

command > *newfile*<**CR**>

in which you can choose to surround the > operator with spaces as indicated in the command line or leave the spaces out (**command>newfile<CR>**); either method is correct.

For example, if you have two files, named **group1** and **group2** each containing a list of names with telephone extension numbers that you would like to sort alphabetically and then interfile into a separate file called **members**, you would type:

sort *group1 group2* > *members*<**CR**>

When you do, the ROS system first alphabetically sorts and then interfiles the contents of the files **group1** and **group2** and redirects the output into the file called **members** rather than displaying it on your terminal. If you wish to read the contents of the **members** file, you could use the **cat** or **pg** command.

Therefore, if the contents of the file **group1** is:

Smith, Allyn	101
Jones, Barbara	203
Cook, Karen	521
Moore, Peter	180
Wolf, Robert	125

and the contents of the file **group2** is:

Frank, M. Jay	118
Nelson, James	210
West, Donna	333
Hill, Charles	256
Morgan, Kristine	175

then the file **members** would appear as follows on your terminal when displayed with the **cat** command.

```
$ sort phase1 phase2 > members<CR>
$ cat members<CR>
Cook, Karen      521
Frank, M. Jay    118
Hill, Charles    256
Jones, Barbara   203
Moore, Peter     180
Morgan, Kristine175
Nelson, James    210
Smith, Allyn     101
West, Donna      333
Wolf, Robert     125
$
```

Keep in mind that if the file to which you are redirecting the standard output already exists, its contents will be replaced with the output of the redirection command.

Redirecting and Appending the Standard Output *(>>)*

Occasionally, you might like to add information to the end of an existing file. You can use the >> operator to do so. Simply input the following command line:

command >> *file*<**CR**>

For example, if the file **members** created in the previous section was subject to additions and deletions, it might be a good idea to date the list so you know at a glance what version of the list you are using. You could do so by typing

date >> members<CR>

on the command line and the date and time would be printed at the end of the file **members**. Or, instead of adding the date to the end of the file **members**, you could have appended another file containing even more names.

Redirecting the Standard Input *(<)*

Standard input can be redirected as well as standard output with the < operator. The general command line format for input redirection is:

command < *file*<CR>

in which the < operator informs the command (or program) to take input from the file you specify rather than from the terminal keyboard (Figure 5-3).

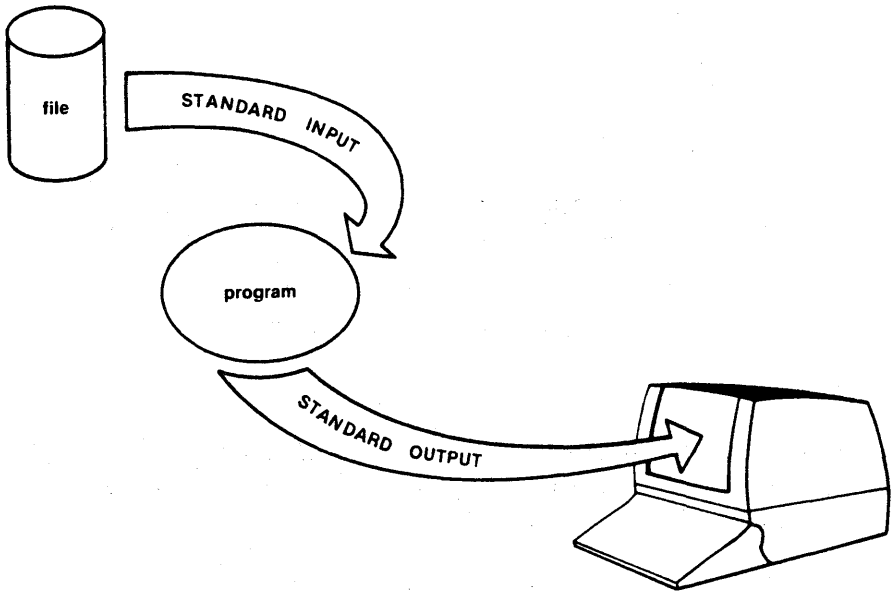


Figure 5-3. Directing Standard Input from File to Program

For example, if you would like to send a copy of the file **members** to co-workers who work on your ROS system and who have the login names **mary2** and **jmrs**, typing

mail mary2 jmrs < members<CR>

will accomplish the task. The **mail** command, however, does not know whether it received its input from the file **members** (which it did) or from your keyboard. Rather, input/output redirection is a service provided by the ROS system shell and is available to every program. (You will learn more about the **mail** command in the *ROS Utility Guide.*)

Connecting Commands with the Pipe (|)

The pipe operator is a powerful and flexible mechanism for doing computing tasks quickly and without the need to develop special purpose tools. You can use it to redirect the standard output of one program to be the standard input of another (Figure 5-4). Generally, the format for using the pipe is:

command | *command*<CR>

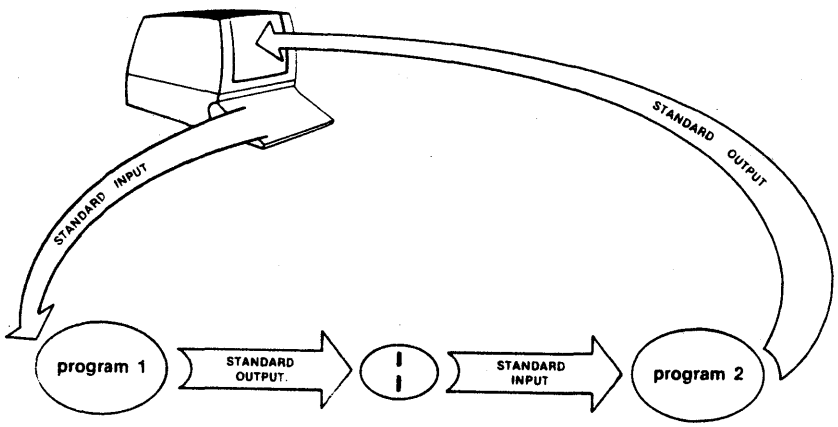


Figure 5-4. Sample Pipe

A popular example of this is taking the output of the **who** command (which you were introduced to in Chapter 3) and using it as input to the **wc** command (which counts lines, words, and/or characters) as follows:

who | wc -l<CR>

This example shows that the standard output of the **who** command was passed to the **wc -l** command (**-l** is the option that counts the number of lines output by the **who** command, each corresponding to a user who is logged into your ROS system.)

Summary

Table 5-3 summarizes which operator performs which redirection task and what general format should be followed in using it. Refer to the section on redirection in Chapter 6 for details on how to use them.

Table 5-3
Options for Redirecting Input and/or Output†

Action	Operator	General Format
Redirecting output to a file	>	command > filename
Redirecting and appending output to a file	>>	command >> filename
Redirecting input from a file	<	command < filename
Redirecting output of first command to be input for second	¦	command ¦ command

RUNNING MULTIPLE PROGRAMS

There are two methods for running multiple programs: you can specify more than one command to execute in sequence from a single command line or you can run commands simultaneously.

Executing Commands in Sequence

Up to this point, the command lines to which you have been introduced and examples for using them have dealt with asking the shell to run a single request or program. For example, each of the command lines **cat filename<CR>**, **date<CR>**, and **ls -l directoryname<CR>** requests the shell to perform one task. You can, however, ask the shell to execute more than one request per command line. Sequential execution allows you to enter as many commands as you wish on one command line and have them execute in the order in which you input them.

To do so, you should first be familiar with the general rules for command line syntax given in Chapter 4. Briefly, command line syntax orders elements in the command line so that the command name, any options you wish to specify, and the data on which the command is to operate (usually the name of a file or directory) follow one another.

To execute more than one command on a line, simply separate the request sequences with semicolons (;) as follows:

command option(s) argument(s); command option(s) argument(s); ...<**CR**>

For example, to determine where you are in the file system and then list the contents of the directory in which you are working, you can type **pwd; ls**<**CR**> and the terminal monitor might read:

```
$pwd; ls<CR>
/user1/starship/bin
dir                 list     tools
$
```

As you can see, the output of the multiple commands is ordered the same way the input is: first, the current working directory is given (in response to **pwd**) and, then, the names of the files and/or directories it holds follow (in response to **ls**).

You could just as easily type:

who am i;date;who<**CR**>

or

mkdir directoryabc;cd directoryabc;pwd<**CR**>

or any combination of commands that you wish to use.

Executing Commands Simultaneously

In addition to running programs sequentially, you can choose to run them simultaneously. To do so, you need to know the difference between foreground and background commands. When a program runs in the background, the computer is executing that program concurrently with the commands that you enter or with the program that you run in the foreground. However, the computer considers your foreground work more important than your background program. This difference has no perceivable effect on the execution of most programs, but running a job in the background is a useful technique when you wish to run a lengthy or time-consuming job without tying up your terminal.

All the command lines used in this guide until now have been examples of foreground commands. This means that they were initiated and run to completion before other commands could be executed and before the shell would return the $ prompt for you to continue. However, you also have the option of running a command in the background so you can continue to work in the foreground.

You can run a command in the background by placing an ampersand (&) at the end of the command line as follows:

command option(s) argument(s) **&<CR>**

When the shell reads the **&**, it starts running the program, prints an identification number, and displays the $ prompt so you can use the terminal immediately for other work.

To save the output from the job you are running in the background, you must redirect the results of the execution into another file so you can look at or use the output when you are ready. For example, if you input the command:

cat file1 file2 > file3 &<CR>

the shell would first give you an identification number, and then the prompt. It will also save the results of **cat file1 file2** in a file named **file3**. When you are ready to peruse **file3**, simply use **cat** or **pg**. If you do not redirect the output, then no output is saved.

When a program is running in the background, it ignores interrupt and break signals, but if you log off, the shell terminates the background program along with the computing session. If you would like to stop a background command while you are still logged into the ROS system, type **kill** *id*<CR>, where *id* is the identification number of the command. On the other hand, to have a program continue to run after you log off, you can use the **nohup** command (which stands for "no hang up") in the following way

nohup command &<CR>

When you do, the command will continue to run until completion and its output is saved in a file called **nohup.out** (which stands for **nohup** output).

CUSTOMIZING YOUR COMPUTING ENVIRONMENT

The information in this section deals with another dimension of control provided to you by the shell called your environment. When you log into the ROS system, the shell automatically sets up a computing environment for you. You can choose to use it as supplied by the system or you can tailor it to meet your needs.

By default, the environment set up by the shell includes the variables:

HOME = your login directory,

PATH = route the shell takes to search for executable files or commands (typically *PATH=:/bin:/usr/bin*), and

LOGNAME = your login name.

If you find the default environment satisfactory, simply leave it as it is and go on with your work. However, if you would like to modify it, you must have a file in your login directory named **.profile**. If you do not, you can create one with a text editor like **ed** or **vi**.

To determine if you have a **.profile**, move to your login directory and type **cat .profile<CR>** and its contents should appear on the terminal monitor. Typically, the **.profile** tests for mail and sets data parameters, system variables, and terminal settings.

Possible modifications to your login environment might include changing your login prompt, setting tab stops, and changing erase and kill characters. If you would like to customize your **.profile**, see the section entitled *MODIFYING YOUR SHELL ENVIRONMENT*, in Chapter 6.

COMMUNICATION UTILITIES

You can send messages or transmit information stored in files to other users who work on your system or on another ROS or UNIX system. If you wish to communicate with remote computers, your ROS system must be physically connected to these computers by means of an Ethernet communications network. The utilities listed in this section offer a variety of methods for transferring files, sending messages, and other types of communication between local computer users or those on remote computers.

LAN -- The Local Area Network (LAN) is an optional hardware/software package that allows you to move files to and from remote systems. LAN also allows you to login onto a remote computer from your local computer. See the *Ridge Local Area Network User's Guide* for details.

Kermit -- **Kermit** allows you to transfer files between computers, or to use one computer as a terminal for the other, on computers with modems. Versions of **kermit** are widely available for different micro- and mini-computers, and some mainframes, allowing you access to a variety of systems. The **kermit** utility is included with the standard ROS software

cu -- The **Cu** (Call Unix) utility allows a terminal on the Ridge 32 to serve as a virtual terminal to most systems requiring RS-232 ASCII terminals. **Cu** allows you to transfer ASCII character data from most other UNIX systems to the Ridge 32, and to other UNIX-compatible systems. The **cu** link is by direct RS-232 connection or telephone modems. **Cu** is standard software with ROS (see *cu*(1) in the *ROS Reference Manual*).

UUCP -- The **UUCP** (Unix-to-Unix Copy) utility copies and transfers files between UNIX systems with direct or modem connections. UUCP is typically used for distribution of software and documentation, personal communication (mail), data transfer, and transmission of debugging dumps. The uucp software manages periodic and automatic contact between the systems on which communications are scheduled. **UUCP** is available on Ridge 32 systems as an add-on feature (see *uucp*(1) in the *ROS Reference Manual*).

mail -- This command is typically used for sending messages to others and reading the messages sent to you. You can use **mail** to send messages or files to other ROS system users using their login names as addresses. And, at your convenience, you can use the **mail** command to read messages sent to you by other users. With **mail**, the recipient can choose when to read it. The **mail** command is described in detail in the *ROS Utility Guide.*

PROGRAMMING IN THE SYSTEM

The ROS system provides an efficient, effective, and convenient environment for programming and software development. This section briefly describes the environment and your programming options when working in it.

If you are not a programmer, your immediate reaction might be to skip this section. But you need not be a programmer or software developer to enjoy some of the capabilities described in this section.

For example, you can use the shell as a command level programming language as well as the command line interpreter. Shell programming capabilities are useful and usable techniques that allow you to take simple, existing programs and make them more powerful. So why not read on.

On the other hand, if you're interested in sophisticated programming and software development capabilities, this section can serve as a springboard to using them.

What you can expect to find in the next few pages is an overview of shell and C language programming and a mention of other languages that can be used on the ROS system. In addition, an overview of some ROS system tools for software development is included.

PROGRAMMING IN THE SHELL

Most interactive users of the ROS system think of the shell solely as the command language interpreter. The shell, however, is also a command level programming language. What this means is that you can let the shell continue to act as your liaison with the computer or you can program the shell to repeat sequences of instructions and to test certain considerations for you automatically. When you program the shell to perform a task, you use the shell to read and to execute commands that you place in an executable file. These files are sometimes called shell scripts or shell procedures.

When you use the shell in this manner, it provides you with features, like variables, control structures, subroutines, and parameter passing that are very similar to those offered by programming languages. These features provide you with the ability to create your own tools by linking together system commands.

For example, you can write a simple shell procedure from existing ROS system programs that tells you the date and time along with the number of users working on your ROS system. One way to do so is illustrated in the following screen:

```
$ cat > users<CR>
date; who | wc -l<CR>
<^d>
$ chmod u+x users<CR>
$
```

A file called **users** is created using the > redirection operator. In the example, **cat** is taking as input everything you type after **<CR>** on the command line and placing it in a file named **users**. Then the file is made executable with the **chmod** command. If you type the command **users<CR>**, your terminal monitor would look something like the next screen.

```
$ users<CR>
Tues May 22 10:29:09 CDT 1984
   7
$
```

The output tells you that seven users were logged into the system when you typed the command at approximately 10:30 A.M. on Tuesday, May 22.

For additional information on shell procedures and for more sophisticated shell programming techniques, see Chapter 6 for step-by-step instructions.

PROGRAMMING IN THE C LANGUAGE

C is a general purpose programming language. C is closely associated with the ROS system because it was developed on the UNIX system and because ROS and UNIX system software is largely written in C.

Although the C programming language is implemented on many computers, it is independent of any particular machine architecture. With a little care, it is easy to write portable programs, that is, programs that can be run without change on a variety of computers if the machine supports a C compiler.

The C programming language comprises the following main elements:

- *Types, operators, and expressions*--Constants and variables are the basic data objects manipulated in a program. Constants are data objects that do not change during the execution of a program, while variables are assigned new values throughout execution. Declarations list variables, state type, and perhaps initial values. Operators specify what is to be done on them. Expressions combine variables and constants to produce new values.
- *Control flow*--Control flow statements of a language specify the order in which computations are done. In C, these include **if-else**, **else-if**, and **switch** statements, and **while**, **for**, and **do-while** loops. In addition, **break**, **continue**, and **goto** statements can be used. Labels can be used as well.
- *Functions and program structure*--C programs generally consist of numerous small functions rather than a few big ones. These functions break large computing tasks into smaller ones and enable you to build on what others have done.
- *Pointers and arrays*--A pointer is a variable that contains the address of another variable. Pointers are frequently used when programming in C because oftentimes they provide the only way to express a computation and partly because their use typically leads to more compact and efficient code than can be obtained in other ways.
- *Structures*--A structure is a collection of one or more variables, possibly of different types, that are grouped together under a single name for convenient handling. Structures help to organize complicated data because they permit a group of related variables to be treated as a unit instead of separate entities.
- *Input and output*--A standard I/O library containing a set of functions designed to provide a standard input and output system is available for C programs. This library is a ROS system feature available for programming in C.

These elements are covered in detail in *The C Programming Language* by B. W. Kernighan and D. M. Ritchie (Prentice-Hall, 1978). Additional information is also available in the *ROS Programmer's Guide.*

OTHER PROGRAMMING LANGUAGES

In addition to C, other programming languages are available for use on the ROS system, such as FORTRAN-77, and Pascal.

You can obtain details on FORTRAN and Pascal in the *ROS Programmer's Guide.*

TOOLS TO AID SOFTWARE DEVELOPMENT

This section highlights some sophisticated software development tools available on the ROS system. The tools are designed to make development of software easier and to provide you with a systematic approach to programming.

There are numerous software development aids that operate with ROS. This section introduces you to five of them to give you an idea of what you can expect development utilities to do. They are:

SCCS -- Source Code Control System

make -- Maintaining programs

lint -- Checks programs for type compliance

lex -- Generating programs for simple lexical tasks

yacc -- Generating parser programs

Refer to the *ROS Utility Guide* and the *ROS Programmer's Guide* for more information.

Source Code Control System *(SCCS)*

The Source Code Control System (SCCS) is a collection of ROS system commands that help you to control and report changes to source code files or text files. SCCS allows you to access different versions of the same file while maintaining only one file. The way this works is that SCCS stores the original file on a disk. Whenever modifications are made to the file, SCCS stores only those changes as a set in something called a **delta**. Each delta or set of changes is numbered to reflect the different versions of a file. You can then choose to retrieve either the original file or a version of the original file.

By allowing SCCS to store and control all iterations of a file, space allocations for storage are minimized and administration of different versions of the same program or document is efficient and simplified. Updates to files can be made quickly and the original version of a program or document is retained if you should need to recall it later.

SCCS is available on Ridge 32 systems as an option. For additional information, see the *ROS Utility Guide.* Most of the commands needed to use SCCS are documented in the *ROS Reference Manual.*

Maintaining Programs *(make)*

The **make** command is a tool for maintaining, supporting, and regenerating large programs or documents on the basis of smaller ones. Since it is easier to handle and modify small programs, it is recommended that if you wish to develop a large program, you start by creating a series of smaller ones that work together to produce the large one.

The **make** command provides you with a method to store all the information you need to assemble small programs or modules into a large, more sophisticated one. A file called a **makefile** holds the file names of the small programs, the steps necessary to generate the large program, and specifies the dependencies among the files.

When **make** executes the **makefile**, the date and time you last modified any of the small programs are checked and the operations needed to update them are performed in sequence. Then, **make** goes on to create the overall large program.

For details on the operation of **make**, see the *ROS Utility Guide.* Or, for a quick reference, see the entry for **make**(1) in the *ROS Reference Manual.*

Checking Programs for Type Compliance *(lint)*

LINT is a C language development tool that checks programs for type compliance and potential portability problems. LINT also detects errors, such as mismatched argument types and uninitialized variables.

For details on the operation of **lint**, see the *ROS Utility Guide.* Or, for a quick reference, see the entry for **lint**(1) in the *ROS Reference Manual.*

Generating Programs for Lexical Tasks *(lex)*

The **lex** utility generates programs to be used in simple lexical analysis of text. Lexical analysis is done by evaluating a stream of characters and constructing the forms that are acceptable to the language. Proper forms are defined in the **lex** program and usable forms can be defined by **lex** defaults or by you. **Lex** produces a subroutine as output that must be compiled and combined with other programs to use the lexical analyzer.

The processing done by the **lex** command can be the first step in creating a compiler-type program. In addition, it can be useful as a preprocessing tool for many different software generation functions.

For additional information on the **lex** command, see the *ROS Utility Guide.* A brief description of how **lex** operates and an explanation of its options can be found in the *ROS Reference Manual.*

Generating Parser Programs *(yacc)*

The **yacc** program, short for **y**et **a**nother **c**ompiler **c**ompiler, is primarily used in the generation of software compilers. Essentially, **yacc** is a utility for creating parser subroutines. The way this works is that first **yacc** uses specified syntax and produces source code for a parser subroutine. Then, the parser subroutine is compiled, and finally used with a program that calls it to parse input. In this way, structure can be imposed on the input to a program and the desired language can be created from defined rules.

See the *ROS Utility Guide* for details on the **yacc** command. Or refer to the *ROS Reference Manual* for some general guidelines on how to use it.

Chapter 6: SHELL TUTORIAL

Chapter 6
SHELL TUTORIAL

INTRODUCTION

You have used the shell to interact with the ROS system by typing in commands that give you information, such as **who**, or commands that perform a task, such as **sort**. This chapter introduces some methods and commands that will help expedite the day-to-day tasks that you perform in the shell.

Two types of shells are available with ROS:

- Bourne Shell
- C-Shell

To avoid confusion, this tutorial will only describe the Bourne shell operations. If, after reading this tutorial, you wish to learn about the C-shell, read the *csh*(1) pages in the *ROS Reference Manual* and the *C-SHELL* section in the *ROS Programmer's Guide.*

The first part of the tutorial, *Shell Command Language*, introduces some basic shortcuts and commands to help you perform tasks in the ROS system quickly and easily. The second part of the tutorial, *Shell Programming*, shows you how to put these tasks into a file and call on the shell to execute the commands in the file while you go get a cup of coffee. The following basics are covered:

- How to use some special characters in the shell,
- How to redirect input and output,
- How to execute and terminate processes,
- How to create and execute a simple shell program,
- How to use variables in a shell program,
- How to use shell programming constructs for looping, conditional execution, and unconditional execution,
- How to locate problems and debug a shell program, and
- How to modify your login environment by editing the file called **.profile**.

HOW TO READ THIS TUTORIAL

Log into your ROS system and try the examples as you read the text. Experiment with the concepts and perhaps combine them into a shell program. Often, there is more then one correct way to write a shell program. You may discover a different method. If your shell program works, if it performs the task, then it is a correct method.

Here is a quick review of the text conventions mentioned at the beginning of this guide.

bold command	(Type in the command line exactly as shown.)
`pica response`	(The system's response to a command.)
italics	(Variable which you or the computer substitutes a name or value)
< >	(Commands that are typed in, but not displayed on your terminal, are enclosed in < >.)
^g	(A control character, hold down the control key CTRL while your press "g".)

A display screen like the one above is used to illustrate the commands and the text of the shell programs. You may not be working on a terminal with a screen. This will not affect the shell tasks that you perform or shell programs that you create. The lines that you type in and the system responses should be the same.

SHELL COMMAND LANGUAGE

SPECIAL CHARACTERS IN THE SHELL

The shell language has special characters that give you some shortcuts for performing tasks in the shell. These special characters are listed below and are discussed in this section of the tutorial.

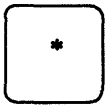
*** ? [ ]** These are metacharacters (also known as "filename expansion characters" or "wildcards"). A metacharacter is a character that has a special meaning in shell command language. These metacharacters give you shortcuts for file names.

&	This character places commands in the background mode. While the shell is performing the commands in the background, your terminal is free for you to work on other tasks.
;	This character allows you to type in several commands on one line. Each command must be followed by a ; . When you type in the <CR>, each command will execute sequentially from the beginning of the line to the end of the line.
\	This character allows you to turn off the meaning of special characters such as *, ?, [ ], & and ; .
" ... " ' ... '	Both double and single quotes turn off the delimiting meaning of the space, and the special meaning of special characters. However, double quotes will allow the characters $ and \ to retain their special meaning. (The $ and \ are discussed later in this chapter and are important for shell programs.

Metacharacters

The meaning of the metacharacters is similar to saying "etc. etc. etc.", "all of the above", or "one of these". Using metacharacters for all or part of a file name is called file name generation. It is a quick and easy way to refer to file names.

Metacharacter That Matches All Characters



This metacharacter matches "all", any string of characters, including no characters at all.

The * alone refers to all the file names in the current directory, the directory you are in now. To see the effect of the *, try the next command.

Type in: **echo *<CR>**

The **echo** command displays its arguments on your terminal. The system response to **echo *** should have been a listing of all file names in the current directory.

Since you may not have used the **echo** command before, here is a brief recap of the command.

Command Recap

echo - write any arguments to the output

command	*options*	*arguments*
echo	none	**any character**
Description:	**echo** writes arguments, which are separated by blanks and ended with <**CR**>, to the output.	
Remarks:	In shell programming, **echo** will be used to issue instructions, to redirect words or data into a file, and to pipe data into a command. All of these uses will be discussed later in this chapter.	

Problem:

Be very careful with * because it is a powerful character. If you type in **rm *** you will erase all the files in your current directory.

The * metacharacter is also used to expand file names in the current directory. If you have written several reports and have named them:

report
report1
report1a
report1b.01
report25
report316

then

report*

refers to all six reports in the current directory. If you want to find out how many reports you have written, you could use the **ls** command to list all the reports that begin with the letters **report**.

```
$ ls report*<CR>
report        report1a      report25
report1       report1b.01   report316
$
```

The * refers to any characters after the letters **report**, including no letters at all.

Notice that * calls the files in numerical and alphabetical order. A quick and easy way to print out all of those reports in order is:

Type in: **pr report*<CR>**

Choose a character that your file names have in common, such as an **a**, and list all those files in the current directory.

Type in: **ls *a*<CR>**

The * can be placed anywhere in the file name.

Type in: **ls F*E<CR>**

This command line would list all of the following files in order:

F123E
FATE
FE
Fig3.4E

Metacharacter That Matches One Character

This metacharacter matches any single character.

The **?** metacharacter replaces any one character of a file name. If you have created text for several chapters of a book, but you only want to list the chapters you have written through **chapter9**, you would use the **?** .

```
$ ls chapter?<CR>
chapter1      chapter2      chapter5
chapter9
$
```

Although **?** matches any one character, you can use it more than once in a file name. To list the rest of the chapters up through **chapter99**, type in:

ls chapter??<CR>

Of course, if you want to list all the chapters in the current directory you would use **chapter***.

Problem:

Sometimes when you **mv** or **cp** a file you accidentally press a character that does not print out on your terminal as part of the file name when you do an **ls**. If you try to **cat** that file, you get an error message. The * and ? are very useful in calling up the file and moving it to the correct name. Try the following example.

1. Make a very short file called **trial**.
2. Type in: **mv trial trial<^g>1<CR>**

   Remember to type in <^g> you hold down the CTRL key and press the "g" key.
3. Type in: **ls trial1<CR>**

```
$ ls trial1<CR>
trial1 not found
$
```

4. Type in: **ls trial?1<CR>**

```
$ ls trial?1<CR>
trial1
$ mv trial?1 trial1<CR>
$ ls trial1<CR>
trial1
$
```

Metacharacters That Match One of a Specific Range of Characters

The shell matches one of the specified characters or range of characters within the brackets.

Characters enclosed in [ ] act as a specialized form of the **?** . The shell will match only one of the characters enclosed in the brackets in the position specified in the file name. If you use **[crf]** as part of a file name, the shell will look for **c**, or **r**, or **f**.

```
$ ls [crf]at<CR>
cat             fat             rat
$
```

The shell will also look for a range of characters within the brackets. For **chapter[0-5]** the shell looks for the files named **chapter0** through **chapter5**. This is an easy way to print out only certain chapters at one time.

Type in: **pr chapter[2-4]<CR>**

This command will print out the contents of **chapter2**, **chapter3**, and **chapter4** in that order.

The shell will also look for a range of letters. For **[A-Z]**, the shell will look for uppercase letters, or for **[a-z]**, the shell will look for lowercase letters.

Try out each of these metacharacters on the files in your current directory.

Commands in the Background Mode

&	**This character, placed at the end of a command line, runs a task in background mode.**

Some shell commands take considerable time to execute. It is convenient to let these commands run in background mode to free your terminal so that you can continue to type in other shell tasks. The general format for a command to run in background mode is:

command &<CR>

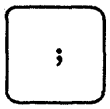
The **grep** command can perform long searches that may take a lot of time. If you place the **grep** command in a background mode, you can continue doing some other task at your terminal while the search is being done by the shell. In the example below, the background mode is used while all the files in the directory are being searched for the characters **word**. The **&** is the last character after the command.

```
$ grep word * &<CR>
21940
$
```

21940 is the process number. This number is essential if you want to stop the execution of a background command. This will be discussed in **Executing and Terminating Processes**.

In the next section of this tutorial you will see how to redirect the system response of the **grep** command into a file so that it does not display on your terminal and interrupt your current work. Then, you can look at the file when you have finished your task.

Sequential Execution



The shell performs sequential execution of commands typed on one line and separated by a ; .

If you want to type in several commands on one line, you must separate each command with a ; . The general format to place **command1**, **command2**, and **command3** on one command line is the following:

command1; command2; command3<CR>

Try out the ; . Type in several commands separated by a ; . Notice that, after you press **<CR>**, the system responds to each command in the order that they appear on the command line.

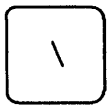
Type in: **cd; pwd; ls; ed trial<CR>**

The shell will execute these commands sequentially:

1. **cd** Change to login directory.
2. **pwd** Print the path of the current directory.
3. **ls** List the files in the current directory.
4. **ed trial** Enter the line editor **ed** and begin editing the file **trial**.

Did you notice the rapid fire response to each of the commands? You may not want these responses to display on your terminal. The section on **Redirecting Output** will show you how to solve this problem.

Turning Off Special Character Meaning

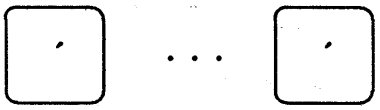


The backslash turns off the special meaning of a metacharacter.

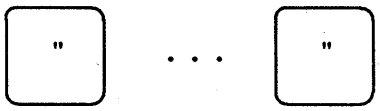
How do you search for one of the special characters in a file? Type in a backslash just before you type in the special character. The backslash turns off the special meaning of the next character that you type in. Create a file called **trial** that has one line containing the sentence "The all * game". Search for the * character in the file **trial**.

```
$ grep \* trial<CR>
The all * game
$
```

Turning Off Special Characters by Quoting



All special characters enclosed in single quotes lose their special meaning.



All special characters except $, \, and ` lose their special meaning when they are in double quotes.

The special characters in the shell lose their special meaning when they are enclosed by quotes. The single quote turns off the special meaning of any character. The double quote will turn off the special meaning of any character except $ and `. The $ and ` are very important characters in shell programming.

A delimiter separates arguments, telling the shell where one argument ends and a new one starts. The space has a special meaning to the shell because it is used as a delimiter between arguments of a command.

The **banner** command uses spaces to delimit arguments. If you have not used the **banner** command, try it out. The system response is rather surprising.

Type in: **banner happy birthday to you<CR>**

Was each word displayed in large poster sized letters?

Now put quotes around **to you**.

Type in: **banner happy birthday "to you"<CR>**

Notice that **to** and **you** appear on the same poster display line. The space between the **to** and the **you** has lost its special meaning as a delimiter.

Since you may not have used the **banner** command before, the following is a quick recap of that command. You may find that you do not have access to the **banner** command. Not all systems have all the commands referenced in this chapter. If you cannot access a command, check with your system administrator.

Command Recap

banner - make posters

command	*options*	*arguments*
banner	none	**characters**
Description:	Displays arguments, up to ten characters on a poster-sized line, in large letters.	
Remarks:	Later in this chapter you will learn how to redirect the **banner** command into a file to be used as a poster.	

If you use single quotes in the argument for the **grep** command, the space loses the meaning of a delimiter. You can search for two words. The line, **The all * game** is in your file **trial**. Look for the two words **The all** in the file **trial**.

```
$ grep 'The all' trial<CR>
The all * game
$
```

Try turning off the special character meaning of the * using single quotes.

```
$ grep '*' trial<CR>
The all * game
$
```

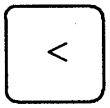
If you want to know more about quoting, read the *sh*(1) pages of the **ROS Reference Manual**.

REDIRECTING INPUT AND OUTPUT

The redirection of input and output are important tools for performing many shell tasks and programs.

Redirecting Input

You can redirect the text of a file to be the input for a command.



This character redirects the contents of a file into a command.

The general format to redirect the contents of a file into a command is shown below.

command < filename<CR>

If you write a report to your boss, you probably do not want to type in the **mail** command and then type in your text. You want to be able to put your report in an editor and correct errors. You want to run the file through the **spell** command to make sure there are no misspelled words. You can **mail** a file containing your report to another login using the input redirection symbol. In the example below, a file called **report** is checked for misspelled words and then redirected to be the input to the **mail** command and mailed to login **boss**.

```
$ spell report<CR>
$ mail boss < report<CR>
$
```

Since the only response to the **spell** command is the prompt, there are no misspelled words in **report**. The **spell** command is a useful tool that gives you a list of words that are not in a dictionary spelling list. The following is a brief recap of **spell**.

Command Recap

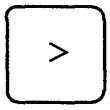
spell - find spelling errors

command	*options*	*arguments*
spell	available*	**filename**
Description:	**spell** collects words from the specified file or files and looks them up in a spelling list. Words that are not on the spelling list are displayed on your terminal.	
Options:	**spell** has several options, including one for checking the British spelling.	
Remarks:	The misspelled words can be redirected into a file. See the redirection symbol > discussed next.	

* See the *ROS Reference Manual* for all available options and an explanation of their capabilities.

Redirecting Output

You can redirect the output of a command to be the contents of a file. When you redirect output into a file, you can either create a new file, append the output to the bottom of a file, or you can erase the contents of an old file and replace it with the redirection output.



This character redirects the output of a command into a file.

The single redirection symbol > will create a new file, or it will erase an old file and replace the contents with new output. The general format to redirect output is shown below.

command > *filename*<CR>

If you want the **spell** command list of misspelled words placed in a file instead of displayed on your terminal, redirect **spell** into a file. In the example, **spell** searches the file **memo** for misspelled words and places those words in the file **misspell**.

```
$ spell memo > misspell<CR>
$
```

The **sort** command can be redirected into a file. Suppose a file called **list** contains a list of names. In the next example, the output of the **sort** command lists the names alphabetically and redirects the list to a new file **names**.

```
$ sort list > names<CR>
$
```

Problem:

Be careful to choose a new name for the file that will contain the alphabetized list. The shell first cleans out the contents of the file that is going to accept the redirected output, then it sorts the file and places the output in the clean file. If you type in

sort list > list<CR>

the shell will erase **list** and then sort nothing into **list**.

Problem:

If you redirect a command into a file that exists, the shell will erase the existing file and put the output of the command into that file. No warning is given that you are erasing an existing file. If you want to assure yourself that there is not an existing file, first execute the **ls** command with the file name as an argument.

If the file exists, **ls** will list the file. If the file does not exist, **ls** will tell you the file was not found in the current directory.

```
$ ls filename<CR>
filename
$ ls junk<CR>
junk not found
$
```

Redirecting Output and Append

The double redirection symbol >> appends the output of a command after the last line of a file.

The general format to append output to a file is:

command >> filename<CR>

In the next example, the contents of **trial2** are added after the last line of **trial1** by redirecting the **cat** command output of **trial2** into **trial1**.

The first command, **cat trial1**, displays the contents of **trial1**. Then, **cat trial2** displays the contents of **trial2**. The third command line, **cat trial2 >> trial1**, adds the contents of **trial2** to the bottom of file **trial1**, and **cat trial1** displays the new contents of **trial1**.

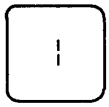
```
$ cat trial1<CR>
hello
this is a trial
This is the last line of this file
$
$ cat trial2<CR>
Add this to file trial1
This is the last line of file trial2
$
$ cat trial2 >> trial1<CR>
$ cat trial1<CR>
hello
this is a trial
This is the last line of this file
Add this to file trial1
This is the last line of file trial2
$
```

In the section on *Special Characters,* one of the examples showed how to execute the **grep** command in background mode with **&**. Now, you can redirect the output of that command into a file called **wordfile**, and then look at the file when you have finished your current task. The **&** is the last character of the command line.

```
$ grep word * > wordfile &<CR>
$
```

Pipes

The ¦ character is called a pipe. It redirects the output of one command to be the input of another command.



This character directs the output from one command to be the input of the next command.

If two or more commands are connected by a pipe, ¦, the output of the first command is "piped" into the next command as the input for that command.

The general format for the pipe line is:

command1 ¦ *command2* ¦ *command3*<**CR**>

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

You have already tried the banner display on your terminal. The pipe can be used to send a banner birthday greeting to someone by electronic mail.

If the person using login **david** has a birthday, pipe the banner display of happy birthday into the **mail** command.

Type in: **banner happy birthday ¦ mail david**<**CR**>

Login **david** will get a banner display in his electronic mail.

The **date** command gives you the date and the time. Since you may not have used the **date** command before, a brief recap of **date** follows.

Command Recap

date - display the date and time

command	*options*	*arguments*
date	**+%m%d%y*** **+%H%M%S**	available*

Description: **date** displays the current date and time on your terminal.

Options: +% followed by m for month, d for day, y for year, H for hour, M for month, and S for second will echo these back to your terminal. You can add an explanation to these such as:

date '+%H:M is the time'

Remarks: If you are working on a small computer system in which you are acting as both user and system administrator, you may be able to set the date and time using optional arguments to the **date** command. Check your reference manual for details. When working in a multiuser environment, the arguments are available only to the system administrator.

* See the *ROS Reference Manual* for all available options and an explanation of their capabilities.

Try out the **date** command on your terminal.

```
$ date<CR>
Mon Nov 25 17:57:21 CST 1985
$
```

Notice that the time is given from the 12th character through the 19th character. If you want to know just the time and not the date, you can pipe the output of the **date** command into the **cut** command. The **cut** command looks for characters only in a specified part of each line of a file. If you use the **–c** option, **cut** will choose only those characters in the specified character positions. Character positions are counted from the left. To display only the time on your terminal, pipe the output of the **date** command into the **cut** command asking for characters 12 through 19.

```
$ date ¦ cut –c12-16<CR>
18:08:23
$
```

Several pipes can be used in one command line. The output of the example can be piped into the **pr** command.

Type in: **date ¦ cut –c12-16 ¦ pr<CR>**

Try each of these examples. Check the system response.

Later in this chapter, you will write a shell program that will give you the time.

Since you may not have used the **cut** command until now, a brief recap of that command follows next.

Command Recap

cut - cut out selected fields of each line of a file

command	*options*	*arguments*
cut	–clist –flist [–d]	**file1 file2**

Description:	**cut** will cut out columns from a table or fields from each line of a file.
Options:	**–c** lists the number of character positions from the left. A range of numbers such as characters 1-9 can be specified by **–c1-9** **–f** lists the number of fields from the left separated by a delimiter described by **–d**. **–d** gives the field delimiter for **–f**. The default is a tab. If the delimiter is a colon, this would be specified by **–d :** .
Remarks:	If you find the **cut** command useful, you may also want to use the **paste** command and the **split** command.

Command Output Substitution

The output of any command line or shell program that is enclosed in back quotes, `, can be substituted anywhere on a shell command line. In the section on *Shell Programming*, you will substitute the output of a command line as the value for a variable.

Substitute the output of the command line in back quotes.

The output of the time command can be substituted for the argument in a banner printout.

Type in: **banner `date ¦ cut –c12-16`<CR>**

Did you get a banner display of the time?

EXECUTING AND TERMINATING PROCESSES

Obtaining the Status of Running Processes

The **ps** command will give you the status of the processes you are running.

Running a process or command in background with & was discussed in the section on special characters. The **ps** command will tell you the status of those processes. In the next example, the **grep** command is run in the background, and then the **ps** command is typed in. The system response, the output from the **ps** command, gives the PID, which is the process identification number, and TTY, which is the current number identification assigned to the terminal you are logged in on. It also gives the cumulative execution TIME for each process, and the COMMAND that is being executed. The PID is an important number if you decide to stop the execution of that command.

```
$grep word * &<CR>
28223
$
$ ps<CR>
ProcessId  Runtime   State Command
22123      4.015       R   /ros/um
23135      0.005       R   /ros/debug
28124      2.416       S   sh
28223      0.015       R   grep
28224      0.002       S   /usr/bin/ps
$
```

The example not only gives you the PID for the **grep** command, but also for the other processes that are running, the **ps** command itself, and the **sh** command (or **csh** command, if using the C-shell) that is always running as long as you are logged in. **sh** is the shell program that interprets the shell commands. It is discussed in Chapter 2 and Chapter 5.

Command Recap

ps - report process status

command	*options*	*arguments*
ps	several*	none
Description:	Displays information about active processes.	
Options:	This command has several options. If you do not use any options you will get the status of the active processes that you are running.	
Remarks:	Gives you the PID, the Process ID. This is needed if you are going to **kill** the process, that is, stop the process from executing.	

* See the *ROS Reference Manual* for all available options and an explanation of their capabilities.

Terminating Active Processes

The **kill** command is used to stop active shell processes. The general format for the **kill** command is:

kill *PID*<**CR**>

What do you do if you decide you do not need to execute the command that you are running in the background?. If you press the BREAK key or the DEL key, you will find it does not stop the background process as it does the interactive commands. The **kill** command terminates a background process. If you want to terminate the **grep** command used in the previous example:

```
$ kill 28223<CR>
28223 Terminated
$
```

If you are running several background processes, you can **kill** all of your background processes by entering: **kill 0**.

It is important to note that the **kill** command normally sends a signal, called signal 15, that *requests* the process to terminate. Some processes have the capability to "catch" or ignore signal 15 and may not terminate right away.

You can unequivocally **kill** a process by specifying **-9** after the **kill** command. This sends a signal 9, which is a *demand*, rather than a *request*, to kill the process. For example, to *demand* that process number 360032 be killed, you would enter:

```
$ kill -9 360032<CR>
360032 Terminated
$
```

and the process will terminate, no matter what.

A recap of the **kill** command follows.

Command Recap

kill - terminate a process

command	*options*	*arguments*
kill	available*	**job number or PID**
Description:	**kill** will terminate the process given by the PID.	

* See the *ROS Reference Manual* for all available options and an explanation of their capabilities.

Using the No Hang Up Command

Another way to kill all processes is to hang up on the system, to log off. What if you want the background process to continue to run after you have logged off? The **nohup** command will allow background commands to continue to run even if you log off.

nohup command &<CR>

If you place the **nohup** command at the beginning of the command that you will be running as a background process, the background process will continue to run to completion after you have logged off.

Type in: **nohup grep word * > word.list &<CR>**

The output will be sent to **word.list**. If you did not specify an output file, the output would be directed to a file named **nohup.out**.

The **nohup** command can be stopped by the **kill** command. The recap of the **nohup** command is the following:

Command Recap

nohup - runs a command, ignoring hanging up or quitting the system

command	*options*	*arguments*
nohup	none	**command line**
Description:	Executes a command line, even if you hang up or quit the system.	

Now that you have mastered these shortcuts in the shell commands, use them in your shell programs.

COMMAND LANGUAGE EXERCISES

1-1. What happens if you use the * at the beginning of a file name? Try to list some of the files in a directory using the * with a last letter of one of your file names. What happens?

1-2. Try out the following two commands.

Type in: **cat [0-9]*<CR>**
echo *<CR>

1-3. Can you use **?** at the beginning or in the middle of a file name generation? Try it.

1-4. Do you have any files that begin with a number? Can you list them without listing the other files in your directory? Can you list only those files that begin with a lowercase letter between a and m? (Hint use a range of numbers or letters in [ ]).

1-5. Can you place a command in background mode on the line that is executing several other commands sequentially. Try it. What happens? (Hint use ; and &.) Can the command in background mode be placed at any position on the command line? Try it. Experiment with each new character that you learn, so that you can learn the full power of the character.

1-6. Using the command line

cd; pwd; ls; ed trial<CR>

redirect the output of **pwd** and **ls** into a file. Remember, if you want to redirect both commands to the same file, you have to use ≫ for the second redirection or you will wipe out the information from the **pwd** command.

1-7. Instead of cutting the time out of the **date** response, try redirecting only the date, without the time, into **banner**. What is the only part that you need to change in the "time" command line?

banner `date ¦ cut –c12-16`

SHELL PROGRAMMING

GETTING STARTED

Let a shell program perform your tasks for you. A shell program is a ROS system file that contains the commands that you would use to perform your task.

- How do you create a simple shell program?
- What makes the program run?
- Is there a special directory for your shell programs?

In this section of the tutorial you will learn the answers to these questions. The examples for creating shell programs usually show two display screens. The first screen displays the contents of the file containing the commands used in your program. It shows the command line

cat file<CR>

and the system response to that command, which is the contents of the file.

```
$ cat file<CR>
First command
    .
    .
    .
Last command
$
```

The $ indicates the Bourne shell prompt. The second screen shows the results of executing your shell program.

```
$ file<CR>
Results
$
```

The names of the file containing the shell program will be printed in **bold** in the text, since it is a command and not an ordinary text file.

Before you begin to create shell programs, you should be familiar with one of the editors. The editors are discussed in the *ROS Text Editing Guide.*

Creating a Simple Shell Program

This section describes how to create a simple shell program that will:

- Tell you the directory you were in,
- List the contents of that directory, and then
- Display on your terminal: "This is the end of the shell program".

To create this shell program, you will need the following three commands:

pwd The command that prints the path name of the current directory,

ls The command that lists the contents of the current directory, and

echo The command that displays on your terminal the characters following **echo**.

Enter an editor and type in the following three commands.

Type in: **pwd<CR>**
ls<CR>
echo This is the end of the shell program.**<CR>**

Write the contents of the editor buffer to a file called **dl** (for directory list) and quit the editor. You have just created a shell program.

```
$ cat dl<CR>
pwd
ls
echo This is the end of the shell program.
$
```

Executing a Shell Program

How do you tell the shell that your file is a shell program that needs to be executed? The simplest way to execute a program is to use the **sh** command.

Type in: **sh dl<CR>**

What happened?

Did you notice the path name of the current directory printed out first, then the list of the contents of the current directory, and last of all the comment *This is the end of the shell program.* ?

The **sh** command is a good way to test out your shell program to make sure that it works.

If **dl** is a useful command, you will want to change the file permissions so that you need only type in **dl** to execute the command. The command that changes the permissions on a file, **chmod**, is discussed in Chapter 4. The example below reminds you how to type in the **chmod** command to make a file executable, and then do an **ls** –**l** so you can see the change in the permissions.

```
$ chmod u+x dl<CR>
$  ls –l<CR>
total 4
–rw------- 1  login login  3661 Nov  2 10:28 mbox
drwxrwxrwx 2  login login  1056 Nov 11 18:20 rje
–rwx------ 1  login login    48 Nov 15 10:50 dl
$
```

Now you have an executable program named **dl** in your current directory.

Type in: **dl<CR>**

Did the **dl** command execute?

Creating a bin Directory for Executable Files

If your shell program is useful, you will want to keep it in a special directory called **bin**, which is under your login directory.

If you want your **dl** command accessible from all your directories, make a **bin** directory from your login directory and move the **dl** file to your **bin**. Below is a reminder of those commands. In this example, **dl** is in the login directory.

Type in: **mkdir bin<CR>**
mv dl bin/dl<CR>

Move to the **bin** directory and type in the **ls** –**l** command. Does **dl** still have execute permission?

Now add the following to the end of the **PATH** entry in your .profile file:

:/*directory path***/bin:**

where *directory path* contains the names of the directories leading to the **/bin** directory. You must then execute the **.profile** file by entering **. .profile**.

For example, if **starship** created the **/bin** directory from his home directory, he would enter **/user1/starship/bin** to the **PATH** entry. Starship's **.profile** file may then look like the following:

```
$ cat .profile
. . .   other entries
. . .
PATH=/bin:/user1/bin:/user1/starship:/user1/games:/user1/starship/bin
$ . .profile
$
```

The **PATH** variable is discussed in more detail in the *Using Shell Variables* section later in this chapter.

After completing these operations, move to another directory other than the **bin** directory and enter:

dl<CR>

The **dl** command is now executable from all of the directories in your account.

A command recap of your new program **dl** follows.

Shell Program Recap

dl - display the directory path and directory contents

command	*arguments*
dl	none
Description:	Displays the output of the shell command **pwd** and then lists the contents of the directory.

The **bin** is the best place to keep your executable shell programs. It is possible to give the **bin** directory another name, but you will need to change the **PATH** variable to specify the new directory name.

Problem:

You can give your shell program file any appropriate file name. However, you should not name your program with the same name as a system command. The system will execute your command instead of the system command.

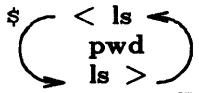
If you had named your **dl** program **mv**, each time you tried to move a file, the system would not move your file. It would have executed your program to display the directory name and list the contents.

Problem:

Another problem would occur if you had named the **dl** file **ls**, and then tried to execute the file **ls**. You would create an infinite loop. After some time, the system would give you an error message:

```
Too many processes, cannot fork
```

What happened? You typed in your new command **ls**. The shell read the command **pwd** and executed that command. Then the shell read the command **ls** in your file and tried to execute your **ls** command. This formed an infinite loop:



echo This is the end of the shell program

ROS limits the number of times this infinite loop can execute. One way to keep this from happening is to give the path name for the system's **ls** command, **/bin/ls**.

The following **ls** shell program would work.

```
$ cat ls<CR>
pwd
/bin/ls
echo This is the end of the shell program
```

If you name your command **ls**, then you can only execute the system command with **/bin/ls**.

VARIABLES

If you enjoyed sending the **banner** birthday greeting, you could make a shell program that would pipe the banner printout into the electronic **mail**. A good shell program would let you send to a different login each time you executed the program. The login would then be a variable. There are two ways you can specify a variable for a shell program:

- Positional parameters and
- Variables that you define.

Positional Parameters

A positional parameter is a variable that is found in a specified position on the command line that invokes your shell program. Positional parameters are typed in after the name of your shell program. They are strings of characters delimited by spaces, except for the last parameter, which is ended with **<CR>**. If **pp1** is the first positional parameter, **pp2** is the second positional parameter, and ... **pp9** is the ninth positional parameter, then the command line that invokes the shell program called **shell.prog** will look like:

shell.prog *pp1 pp2 pp3 pp4 pp5 pp6 pp7 pp8 pp9*<CR>

The shell program will take the first positional parameter (**pp1**) and substitute it in the shell program text for the characters **$1**. The second positional parameter (**pp2**) will be substituted for the characters **$2**. The ninth positional parameter (**pp9**), of course, will be substituted for the characters **$9**.

If you want to see how the positional parameters are substituted into a program, try typing the following lines into a file called **pp** (positional parameters).

Type in:
```
echo The first positional parameter is: $1<CR>
echo The second positional parameter is: $2<CR>
echo The third positional parameter is: $3<CR>
echo The fourth positional parameter is: $4<CR>
```

First the **echo** command tells which parameter will be displayed and then displays the parameter. The next example shows the contents of the file **pp**.

```
$ cat pp<CR>
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
$
```

The following example shows the results of giving the four positional parameters one, two, three, and four to the shell program **pp**. Remember to change the mode of **pp** to be executable.

```
$ chmod u+x pp<CR>
$
$ pp one two three four<CR>
The first positional parameter is: one
The second positional parameter is: two
The third positional parameter is: three
The fourth positional parameter is: four
$
```

Now, return to creating your shell program for the banner birthday greeting. Call the file **bbday**. What command line would go into that file? Before you go on reading, try it.

Did you get the following?

```
$ cat bbday<CR>
banner happy birthday | mail $1
```

Try sending yourself a birthday greeting. If your login name is **slowmo**, then the command line would be:

```
$ bbday slowmo<CR>
you have mail
$
```

The following is a brief recap of the shell program command **bbday**.

Shell Program Recap

bbday - mail a banner birthday greeting

command	*arguments*
bbday	**login**
Description:	**bbday** mails "happy birthday" in poster-sized letters to the specified login.

The **who** command will tell you every login that is currently using the system. How would you make a simple shell program called **whoson** that will tell you if a particular login is currently working on the system? You could try the following:

```
$ who ¦ grep boss<CR>
boss   tty51 Nov 29 17:01
$
```

This command pipes the output of the **who** command into the **grep** command. The **grep** command is searching for the characters "boss". Since login **boss** is currently logged into the system, the shell will respond with:

boss tty51 Nov 29 17:01

If the only response is a prompt sign, then login **boss** is not currently on the system because the **grep** command found nothing. Create a **whoson** shell program.

Below are the ingredients for your shell program **whoson**.

who The shell command that lists everyone on the system,

grep The search command, and

$1 The first positional parameter for your shell program.

The **grep** command searches the output of the **who** command for the parameter designated in the program by **$1**. If it finds the login, it will display the line of information. If it does not find the login in the output from **who**, it will display your prompt.

Enter an editor and type the following command line into a file called **whoson**.

Type in: **who ¦ grep $1<CR>**

Write the file, quit the editor, and use the **chmod** command to give execute permission to the file **whoson**.

Now try using your login as the positional parameter for the new program **whoson**. What was the system's response?

If your login name is **slowmo**, your new shell command line would look like:

```
$ whoson slowmo<CR>
slowmo    tty26        Jan 24 13:35
$
```

The first positional parameter is **slowmo**. The shell substitutes **slowmo** for the **$1** in your program.

who ¦ grep slowmo<CR>

The following is a brief recap of the **whoson** command.

Shell Program Recap

whoson - display login information if user is logged in

command	*arguments*
whoson	**login**
Description:	If a user is on the system, displays the user's login, the **TTY** number, the time and date the user logged in.

The shell command line will allow 128 positional parameters. However, your shell program text is restricted to **$1** through **$9**, unless you use the **$*** described below.

Parameters with Special Meaning

$# This variable in your shell program will record and display the number of positional parameters you typed in for your shell program.

Let's look at an example that will show you what happens when you use **$#**. Put the following command lines in a shell program called **get.num**.

```
$ cat get.num<CR>
echo The number of parameters is: $#
$
```

The program counts all the positional parameters and displays that number. Give **get.num** four parameters. They can be any string of characters.

```
$ get.num test out this program<CR>
The number of parameters is: 4
$
```

Shell Program Recap

get.num - count and display the number of arguments

command	*arguments*
get.num	(any string)
Description:	**get.num** counts the number of arguments given to the command and then displays that number.
Remarks:	This command demonstrates the special parameter $#.

$* This variable in your shell program will substitute all positional parameters starting with the first positional parameter. The parameter $* does not restrict you to nine parameters.

You can make a simple shell program to demonstrate $*. Make a shell program called **show.param** that will **echo** all of the parameters. Type in the **echo** command line shown in the following screen.

```
$ cat show.param<CR>
echo The parameters for this command are: $*
$
```

Make **show.param** executable and try it out.

```
$ show.param hello how are you<CR>
The parameters for this command are: hello how are you
$
```

Now try **show.param** using more than nine positional parameters.

```
$ show.param one two 3 4 5 six 7 8 9 10 11<CR>
The parameters for this command are: one two 3 4 5 six
7 8 9 10 11
$
```

The $* is very handy if file generation names are used as the parameters.

Try a file name generation parameter in your **show.param** command. If you have several chapters of a manual in your directory called chap1, chap2 through chap7, you will get a printout listing of all of those chapters.

```
$ show.param chap?<CR>
The parameters for this command are: chap1 chap2 chap3
chap4 chap5 chap6 chap7
$
```

A quick recap of **show.param** follows.

Shell Program Recap

show.param - display all of the parameters

command	*arguments*
show.param	(any positional parameters)
Description:	**show.param** displays all of the parameters.
Remarks:	If the parameters are file name generations, it will display each of those file names.

You may want to practice with positional parameters so that they are familiar to you before you continue on to the next section in which you will name the variables within the program, rather than use them as arguments in a command line.

Variable Names

The shell allows you to name the variables within a shell program. Naming the variables in a shell program makes it easier for another person to use. Instead of using positional parameters, you will tell the user what to type in for the variable, or you will give the variable a value that is the output of a command.

What does a named variable look like? In the example below, **var1** is the name of the variable and **myname** is the value or character string assigned to that variable. There are no spaces on either side of the = sign.

var1=myname<CR>

Within the shell program, a $ in front of the variable name alerts the shell that a substitution is needed in the shell program. **$var1** tells the shell to substitute the value **myname**, which was given to **var1**, for the characters **$var1**.

The first character of a variable name must be a letter or an underscore. The rest of the name can be composed of letters, underscores, and digits. As in the case of shell program file names, it is a risky business to use a shell command as a variable name. Also, the shell has reserved some variable names to be used by the shell. The following names are used by the shell and should not be used as the name of one of your variables. A brief explanation of each variable is given.

CDPATH
This variable defines the search path for the **cd** command.

HOME
This is the default variable for the **cd** command (Home Directory).

IFS
This variable defines the internal field separators, normally the space, the tab, and the carriage return.

MAIL
This variable is set to the name of the file that contains your electronic mail.

PATH
This variable determines the path that is followed to find commands.

PS1
PS2
These variables define the primary and secondary prompt strings. The defaults are $ and >. Do you have a prompt sign $?

TERM
This variable tells the shell what kind of terminal you are working on. It is important to set this variable if you are editing with **vi**.

Many of these named variables are explained in the last section of this chapter on your login environment.

Assign Values to Variables

If you edit with **vi**, you know that you must set the variable TERM to equal the code for your type of terminal before you use the **vi** editor. For example:

TERM=T3<CR>

This is the simplest way to assign a value to a variable.

There are several other ways to assign values to variables. One way is to use the **read** command to assign input to the variable. Another way is to assign the value from the output of a command using back quotes ` ... `. A third way would be to assign a positional parameter to the variable.

Assign Values by the Read Command

You can set up your program so that you can type in the command and then be prompted by the program to type in the value for the variable. The **read** command assigns the input to the specified variable. The general format for the **read** command is:

read *var*<CR>

The values assigned by **read** to **var** will be substituted for **$var** in the program. If the **echo** command is executed just before the **read** command, the program can display the directions "type in ...". The **read** command will wait until you type in the value, and then assign the string of characters that you type in as the value for the variable.

If you had a list that contained the names and telephone numbers of people you called often, you could make a simple shell program that would automatically give you someone's number. Stop for a minute. How would you make up the program using the following ingredients?

echo The command that echoes the instructions.

read The command that assigns the input value to the variable **name**.

grep The command that searches for the person's name and number.

First, you would use the **echo** command to inform the user to type in the name of the person to be called.

echo *person's last name*<**CR**>

The **read** command will then assign the person's name to the variable **name**.

read name<**CR**>

Notice that you do not use the = to assign the variable, the **read** command automatically assigns the typed in characters to **name**.

The **grep** command will then search your phone list for the name. If your phone list were called **list**, the command line would be:

grep $name list<**CR**>

In the next example, the shell program is called **num.please**. Remember, the system response to the **cat** command is the contents of the shell program file.

```
$ cat num.please<CR>
echo Type in the last name
read name
grep $name list
$
```

Make a list of last names and phone numbers and try **num.please**. Or, try the next example, which is a program that creates a list. You can use several variables in one program. If you have a phone list, you may want a quick and easy way to add names and numbers to the list. The program:

- Asks for the name of the person,
- Assigns the name to the variable **name**,
- Asks for the person's number,
- Assigns the number to the variable **num**, and
- Echos the **name** and **num** into the file **list**. You must use ≫ to redirect the output of the **echo** command to the bottom of your list. If you use >, your list will contain only the last phone number.

The program is called **mknum**.

```
$ cat mknum<CR>
echo Type in name
read name
echo Type in number
read num
echo $name $num >> list
$
$ chmod u+x mknum<CR>
$
```

Now try out the new programs for your phone list. In the next example, **mknum** creates the new listing for Mr. Niceguy. Then, **num.please** gives you Mr. Niceguy's phone number.

```
$ mknum<CR>
Type in the name
Mr. Niceguy<CR>
Type in the number
668-0007<CR>
$
$ num.please<CR>
Type in last name
Niceguy<CR>
Mr. Niceguy 668-0007
$
```

Notice that the variable **name** accepts both **Mr.** and **Niceguy** as the value.

Here is a brief recap of **mknum** and **num.please**.

Shell Program Recap

mknum - place name and number on a phone list

command	*arguments*
mknum	(interactive)
Description:	Asks you for the name and number of a person and adds the name and number to your phone list.
Remarks:	This is an interactive command.

Shell Program Recap

num.please - display a person's name and number

command	*arguments*
num.please	(interactive)
Description:	Asks you for a person's last name, and then displays the name and telephone number.
Remarks:	This is an interactive command.

Substitute Command Output for the Value of a Variable

Another way to assign a value to a variable is to substitute the output of a command for the value. This will be very useful in the next section when you try loops and conditional constructs.

The general format to assign output as the value for a variable is:

var=`** *command***` **<CR>**

The variable **var** has the value of the output from **command**.

In one of the previous examples on piping, the **date** command was piped into the **cut** command to get the correct time. That command line was:

date | cut –c12-19<CR>

You can place that command in a simple shell program called **t** that will give you the time.

```
$ cat t<CR>
time=`date | cut -c12-16`
echo The time is: $time
$
```

Remember there are no spaces on either side of the equal sign.

Change the mode on the file and you now have a program that gives you the time.

```
$ chmod u+x t<CR>
$ t<CR>
The time is: 10:36
$
```

The recap for the **t** shell program follows.

Shell Program Recap

t - display the correct time

command	*arguments*
t	none
Description:	**t** gives you the correct time in hours and minutes.

Assign Values with Positional Parameters

A positional parameter can be assigned to a named parameter. For example:

var1=$1<CR>

The example below is a simple program **simp.p** that demonstrates how you can assign a positional parameter to a variable. The command lines in the file would be the following:

```
$ cat simp.p<CR>
var1=$1
echo $var1
$
```

Or, you can assign the output of a command that uses a positional parameter.

person=`who ¦ grep $1`<CR>

If you wanted to keep track of the results of your **whoson** program, you could create the program **log.time**. The output of your **whoson** shell program is assigned to the variable **person**. Then, that value **$person** is added to the file **login.file** with the

echo command. The last part of the program displays the value of **$person**, which is the same as the response to the **whoson** command.

```
$ cat log.time<CR>
person=`who | grep $1`
echo $person >> login.file
echo $person
$
```

The system response to **log.time** would appear as in the following screen.

```
$ log.time maryann<CR>
maryann     tty61          Apr 11 10:26
$
```

The following is a quick recap of the **log.time** program.

Shell Program Recap

log.time - log and display a specified login that is currently logged in

command	*arguments*
log.time	**login**

Description: If the specified login is currently on the system, **log.time** places the line of information from the **who** command into the file **login.file** and then displays that line of information on your terminal.

As you do more programming, you may discover other ways to assign variables that will help you in shell programs.

SHELL PROGRAMMING CONSTRUCTS

The shell programming language has several constructs that give you more flexibility in your programs.

- The "here document" allows you to redirect lines of input into a command.
- The looping constructs **for** or **while** cause a program to reiterate commands in a loop.

- The conditional control commands, **if** or **case**, execute a group of commands only if a particular set of conditions is met.
- The **break** command gives the unconditional end of a loop.

Comments

Before you begin writing shell programs with loops, you may want to know how to put comments about your program into the file, which the system will ignore. To place comments in a program, begin the comment with # and end it with <CR>. The general format for a comment line is:

#comment<CR>

The shell will ignore all characters after the #. These lines

 # This program sends a generic birthday greeting<CR>
 # This program needs a login as the positional parameter<CR>

will be ignored by the system when your program is being executed. They only serve as a reminder to you, the programmer.

The Here Document

The here document allows you to redirect lines of input of a shell program into a command. The here document consists of the redirection symbol << and the delimiter that specifies the beginning and end of the lines of input. The delimiter can be one character or a string of characters. The ! is often used as a delimiter. The general format for the here document is:

command <<!<CR>
...input lines...<CR>
!<CR>

The here document could be used in a shell program, to redirect lines of input into the **mail** command. The program shown below sends a generic birthday greeting with the **mail** command. The program is called **gbday**.

```
$ cat gbday<CR>
mail $1 <<!
Best wishes to you on your birthday.
!
$
```

The person's login is the first positional parameter **$1**.

The redirected input is:

Best wishes to you on your birthday.

To send the greeting:

```
$ gbday mary<CR>
$
```

To receive the greeting, login **mary** would execute the **mail** command.

```
$ mail<CR>
mailx version 2.14 06/08/85  Type ? for help.
"/usr/mail/mary": 1 message 1 new
>N  1 mylogin      Wed Jul 17 14:26   11/281
? <CR>
Message  1:
From mylogin Wed Jul 17 10:19 PDT 1985
Received: by system names (1.4/4.7)
       id AA2818325; Wed, 17 Jul 85 10:19:07 pdt
Date: Wed, 17 Jul 85 10:19:07 pdt
From: mylogin (My Name)
Message-Id: <8507171719.AA2818325>
To: mary
Status: RO

Best wishes to you on your birthday

? q<CR>
Saved 1 message in /usr/mary/mbox
```

The following is a recap of **gbday**.

Shell Program Recap

gbday - send a generic birthday greeting

command	*arguments*
gbday	**login**
Description:	**gbday** sends a generic birthday greeting to the login given as an argument.

Using ed in a Shell Program

The line editor **ed** can be used within a shell program if it is combined with the here document commands.

Suppose you want to make a shell program that will enter the editor, **ed**, make a global substitution to a file, write the file, and then quit the editor. The **ed** command to make a global substitution is:

g/text to be changed/s//new text/g<CR>

Before you read any further, jot down what you think the command sequence will be. Put your command sequence into a file called **ch.text**. If you want to suppress the character count of **ed** so that it will not appear on your terminal, use the – option:

ed – *filename*<**CR**>

Try to execute the file. Did it work?

If you used the **read** command to enter the variables, your program **ch.text** may look similar to what appears in the following screen.

```
$ cat ch.text<CR>
echo Type in the file name.
read file1
echo Type in the exact text to be changed.
read oldtext
echo Type in the exact new text to replace the above.
read newtext
ed - $file1 <<!
g/$oldtext/s//$newtext/g
w
q
!
$
```

This program uses three variables. Each of them is entered into the program with the **read** command.

$file The name of the file to be edited.

$oldtext The exact text to be changed.

$newtext The new text.

Once the variables are entered into the program, the here document redirects the global, write, and quit commands into the **ed** command. Try out the new **ch.text** command.

```
$ ch.text<CR>
Type in the filename.
memo<CR>
Type in the exact text to be changed.
Dear John:<CR>
Type in the exact new text to replace the above.
To whom it may concern:<CR>
$ cat memo<CR>
To whom it may concern:
$
```

Did you try to use positional parameters? Did you have any problems entering the text changes as variables, or did you quote the character strings for each parameter?

The recap of the **ch.text** command is:

Shell Program Recap

ch.text - change text in a file

command	*arguments*
ch.text	(interactive)
Description:	Replaces text in a file with new text.
Remarks:	This shell program is interactive. It will prompt you to type in the arguments.

If you want to become more familiar with the line editor **ed**, or any of the other editors that run with ROS, see the *ROS Text Editing Guide.*

Looping

Until now, the commands in your shell program have been executed once and only once and in sequence. Looping constructs give you repetitive execution of a command or group of commands. The **for** or **while** constructs will cause a program to loop and execute a sequence of commands several times.

The for Loop

The **for** loop executes a sequence of commands for each member of a list. The **for** command loop also requires the keywords **in, do,** and **done**. The **for, do,** and **done** keywords must be the first word on a line. The general format of the **for** loop is:

```
for variable<CR>
      in this list of values<CR>
do the following commands<CR>
     command 1<CR>
     command 2<CR>
          .
          .
          .
     last command<CR>
done<CR>
```

The variable can be any name you choose. If it is **var**, then the values given after the keyword **in** will be sequentially substituted for **$var** in the command list. If **in** is omitted, the values for **var** will be the positional parameters. The command list between the keywords **do** and **done** will be executed for each value.

When the commands have been executed for the last value, the program will execute the next line below **done**. If there is no line, the program will end.

It is easier to read a shell program if the looping constructs stand out. Since the shell ignores spaces at the beginning of the lines, each section of commands can be indented as it was in the above format. Also, if you indent each command section, you can quickly check to make sure each **do** has a corresponding **done** statement to end the loop.

The easiest way to understand a shell programming construct is to try an example. Try to create a program that will move files to another directory.

The ingredients for the program are:

echo	You want to echo directions to type in the path name to reach the new directory.
read	You want to type in the path name, and assign it to the variable **path**.

for *variable*	You must name the variable. Call it **file** for your shell program. It will appear as **$file** in the command sequence.
in *list of values*	The list of values will be the file names. If the **in** clause is omitted, the list of values is taken to be **$***, that is, the parameters entered on the command line.
do *command sequence*	The command sequence for this program is: **mv $file $path/$file<CR>**
done	

Your shell program text for the program called **mv.file** will look like:

```
$ cat mv.file<CR>
echo Please type in the directory path
read path
for file
   in memo1 memo2 memo3
do
   mv $file $path/$file
done
$
```

Notice that you did not type in any values for the variable **file**. The values are already in your program. If you want to change the files each time you invoke the program, use positional parameters or variables that you name. You do not need the **in** keyword to list the values when you use positional parameters. If you choose positional parameters, your shell program will look like:

```
$ cat mv.file<CR>
echo type in the directory path
read path
for file
do
   mv $file $path/$file
done
$
```

It is likely that you will want to move several files using the special file name generation characters.

If this is a useful command, remember to move it into your **bin**.

Following is a recap of the **mv.file** shell program.

Shell Program Recap

mv.ex - move all executable files in the current directory to the **bin** directory

command	*arguments*
mv.ex	**all file names (*)**
Description:	Moves all the files with execute permission that are in the current directory to the **bin** directory
Remarks:	All executable files in the **bin** directory (or the directory indicated by the **PATH** variable) can be executed from any of your directories.

The while Loop

The **while** loop will continue executing the sequence of commands in the **do...done** list as long as the final command in the **while** command list returns a status of true, that is can be executed. The **while, do,** and **done** keywords must be the first characters on the line. The general format of the **while** loop is the following:

```
while<CR>
      command 1<CR>
            .
            .
            .
      last command<CR>
do<CR>
      command 1<CR>
            .
            .
            .
      last command<CR>
done<CR>
```

A simple program using the **while** loop enters a list of names into a file. The command lines for that program called **enter.name** are:

```
$ cat enter.name<CR>
while
   read x
do
   echo $x>>xfile
done
$
```

This shell program needs some instructions. You have to know to delimit or separate the names by a **<CR>**, and you have to use a **<^d>** to end the program. Also, it would be nice if your program displayed the list of names in the **xfile** at the end of the program. If you added those ingredients to the program, the commands lines for the program become:

```
$ cat enter.name<CR>
echo 'Please type in each person's name and then a <CR>'
echo 'Please end the list of names with a <^d>'
while read x
do
   echo $x>>xfile
done
echo xfile contains the following names:
cat xfile
$
```

Notice that after the loop is completed, the program executes the commands below the **done**.

In the **echo** command line, you used characters that are special to the shell, so you must use the **'...'** to turn off that special meaning. Put the above command lines in an executable file and try out the shell program.

```
$ enter.name<CR>
Please type in each person's name and then a <CR>
Please end the list of names with a <^d>
Mary Lou<CR>
Janice<CR>
<^d>
xfile contains the following names:
Mary Lou
Janice
$
```

Conditional Constructs if...then

The **if** command tells the shell program to execute the **then** sequence of commands only if the final command in the **if** command list is successful. The **if** construct ends with the keyword **fi**. The general format for the **if** construct is as follows:

```
if<CR>
    command1<CR>
        .
        .
        .
    last command<CR>
  then<CR>
      command1<CR>
          .
          .
          .
      last command<CR>
fi<CR>
```

The next shell program demonstrates the **if...then** construct. The program will search for a word in a file. If the **grep** command is successful then the program will **echo** that the word is found in the file. In this example the variables are read into the shell program. Type in the shell program shown below and try it out. Call the program **search**.

```
$ cat search<CR>
echo Type in the word and the file name.
read word file
if grep $word $file
   then echo $word is in $file
fi
$
```

Notice that the **read** command is assigning values to two variables. The first characters that you type in, up to a space, are assigned to **word**. All of the rest of the characters including spaces will be assigned to **file**.

Pick a word that you know is in one of your files and try out this shell program. Did you see that even though the program works, there is an irritating problem? Your program displayed more than the line of text called for by the program. The extra lines of text displayed on your terminal were the output of the **grep** command.

The Shell Garbage Can /dev/null

The shell has a file that acts like a garbage can. You can deposit any unwanted output in the file called **/dev/null**, by redirecting the command output to **/dev/null**.

Try out **/dev/null** by throwing out 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 response into the file **/dev/null**.

who > /dev/null<CR>

The response displayed on your terminal was your prompt. The response to the **who** command was placed in **/dev/null** and became null, or nothing. If you want to dispose of the **grep** command response in your **search** program, change the **if** command line.

if grep $word $file > /dev/null<CR>

Now execute your **search** program. The program should only respond with the text of the **echo** command line.

The **if...then** construction can also issue an alternate set of commands with **else**, when the **if** command sequence is false. The general format of the **if...then...else** construct follows.

```
if<CR>
     command1<CR>
          .
          .
          .
     last command<CR>
   then<CR>
        command1<CR>
             .
             .
             .
        last command<CR>
   else<CR>
        command1<CR>
             .
             .
             .
        last command<CR>
fi<CR>
```

You can now improve your **search** command. The shell program **search** can look for a word in a file. If the word is found, the program will tell you the word is found. If it is not found (**else**) the program will tell you the word was NOT found. The text of your **search** file will look like the following:

```
$ cat search<CR>
echo Type in the word and the file name.
read word file
if
   grep $word $file >/dev/null
then
   echo $word is in $file
else
   echo $word is NOT in $file
fi
$
```

Following is a quick recap of the enhanced shell program called **search**.

Shell Program Recap

search - tell if a word is in a file

command	*arguments*
search	interactive
Description:	Tells the user whether or not a word is in a file.
Remarks:	The arguments, the word and the file, are asked for interactively.

The test Command for Loops

test is a very useful command for conditional constructs. The **test** command checks to see if certain conditions are true. If the condition is true, then the loop will continue. If the condition is false, then the loop will end and the next command is executed. Some of the useful options for the **test** command are:

test **–r** *filename*<**CR**>
True if the file exists and is readable

test **–w** *filename*<**CR**>
True if the file exists and has write permission

test **–x** *filename*<**CR**>
True if the file exists and is executable

test **–s** *filename*<**CR**>
True if the file exists and has at least one character

If you have not changed the values of the **PATH** variable that were initially given to you by the system, then the executable files in your **bin** directory can be executed from any one of your directories. You may want to create a shell program that will move all the executable files in the current directory to your **bin** directory. The **test –x** command can be used to select the executable files from the list of files in the current directory. Review the **mv.file** program example of the **for** construct.

```
$ cat mv.file<CR>
echo type in the directory path
read path
for file
do
  mv $file $path/$file
done
$
```

Include an **if test** **-x** statement in the **do...done** loop to move only those files that are executable.

If you name the shell program **mv.ex**, then the shell program will be as follows:

```
$ cat mv.ex<CR>
echo type in the directory path
read path
for file
  do
    if test -x $file
       then
         mv $file $path/$file
    fi
  done
$
```

The directory path will be the path from the current directory to the **bin** directory. However, if you use the value for the shell variable **HOME**, you will not need to type in the path each time. **$HOME** gives the path to the login directory. **$HOME/bin** gives the path to your **bin**.

```
$ cat mv.ex<CR>
for file
  do
    if test -x $file
       then
         mv $file $HOME/bin/$file
    fi
  done
$
```

To execute the command, use all the files in the current directory, * , as the positional parameters. The following screen executes the command from the current directory and then moves to the **bin** directory and lists the files in that directory. All the executable files should be there.

```
$ mv.ex *<CR>
$ cd; cd bin; ls<CR>
```

The Conditional Construct case..esac

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

```
case characters<CR>
in<CR>
    pattern1)<CR>
        command line 1<CR>
            .
            .
            .
        last command line<CR>
    ;;<CR>
    pattern2)<CR>
        command line 1<CR>
            .
            .
            .
        last command line<CR>
    ;;<CR>
    pattern3)<CR>
        command line 1<CR>
            .
            .
            .
        last command line<CR>
    ;;<CR>
    *)<CR>
        command 1<CR>
            .
            .
            .
        last command<CR>
    ;;<CR>
esac<CR>
```

The **case** construction will try to match characters with the first pattern. If there is a match, the program will execute the command lines after the first pattern and up to the ;;.

If the first pattern is not matched, then the program will proceed to the second pattern. After a pattern is matched, the program does not try to match any more of the patterns, but goes to the command following **esac**. The * used as a pattern at the end of the list of patterns allows you to give instructions if none of the patterns are matched. The * means any pattern, so it must be placed at the end of the pattern list if the other patterns are to be checked first.

If you have used the **vi** editor, you know you must assign a value to the **TERM** variable so that the shell knows what kind of terminal is going to display the editing window of **vi**. A good example of the **case** construction would be a program that will set up the shell variable **TERM** for you according to what type of terminal you are logged in on. If you log in on different types of terminals, the program **set.term** will be very handy for you.

set.term will ask you to type in the terminal type, then it will set **TERM** equal to the terminal code. You may want to glance back at the beginning of the **vi** tutorial for the explanation of those commands. The command lines are:

TERM=terminal code<**CR**>
export TERM<**CR**>

In this example of **set.term**, the person uses either a Ridge Graphics Display or a Televideo 950.

The **set.term** program will first check if the value of **term** is **ridge**. If it is, then it will assign the value **ridge** to **TERM**, and exit the program. If it is not **ridge**, it will check for **televideo**. It will execute the commands under the first pattern that it finds, and then go to the next command after the **esac** command.

At the end of the patterns for the terminals, the pattern * , meaning everything else, will warn you that you do not have a pattern for that terminal, and it will also allow you to leave the **case** construct.

```
$ cat set.term<CR>
echo If you have a Ridge Display type in ridge
echo If you have a Televideo 950 type in televideo
read term
case $term
  in
     ridge)
        TERM=ridge
     ;;
     televideo)
        TERM=tvi950
     ;;
     *)
        echo not a correct terminal type
     ;;
esac
export TERM
echo end of program
$
```

What would have happened if you had placed the * pattern first? The **set.term** program would never assign a value to TERM since it would always fit the first pattern *, which means everything.

When you read the section on modifying your login environment, you may want to put the **set.term** command in your **bin**, and add the command line

set.term<CR>

to your **.profile.**

Following is a quick recap of the **set.term** shell program.

Shell Program Recap

set.term - assign a value to TERM

command	*arguments*
set.term	interactive

Description:	Assigns a value to the shell variable TERM and then exports that value to other shell procedures.
Remarks:	This command asks for a specific terminal code to be used as a pattern for the **case** construction.

Unconditional Control Statement break

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 the program **set.term**, the **break** command could have been used instead of the **echo** command.

```
$ cat set.term<CR>
echo If you have a Ridge Display type in ridge
echo If you have a Televideo 950 type in televideo
read term
case $term
   in
      ridge)
         TERM=ridge
      ;;
      televideo)
         TERM=tvi950
      ;;
   *)
      break
   ;;
esac
export TERM
echo end of program
$
```

As you do more shell programming, you may want to use two other unconditional commands, the **continue** command and the **exit** command. The **continue** command causes the program to go immediately to the next iteration of a **do** or **for** loop without executing the remaining commands in the loop.

Normally, a shell program terminates when the end of the file is reached. If you want the program to end at some other point, you can use the **exit** command.

DEBUGGING PROGRAMS

Debugging is computer slang for finding and correcting errors in a program. There will be times when you will execute a shell program and nothing will happen. There is a "bug" in your program.

Your program may consist of several steps or several groups of commands. How do you discover which step is the culprit? There are two options to the **sh** command that will help you debug a program.

sh –v<CR>	Prints the shell input lines as they are read by the system.
sh –x<CR>	Prints commands and their arguments as they are executed.

To try out these two options, create a shell program that has an error in it. For example, type in the following list of commands in a file called **bug**.

```
$ cat bug<CR>
today=`date`
person=$1
mail $2
$person
When you log off come into my office please.
$today.
MLH
$
```

The mail message sent to Tom ($1) at login **tommy** ($2) should read as shown in the following screen.

```
$ mail<CR>
mailx version 2.14 06/08/85  Type ? for help.
"/usr/mail/tommy": 1 message 1 new
>N  1 mlh      Wed Jul 17 14:26   11/281
? <CR>
Message  1:
From mlh Wed Jul 17 10:19 PDT 1985
Received: by system names (1.4/4.7)
        id AA3815328; Wed, 17 Jul 85 10:19:07 pdt
Date: Wed, 17 Jul 85 10:19:07 pdt
From: mlh (Melvin Hopper)
Message-Id: <8507171719.AA3815328>
To: tommy
Status: RO

Tom
When you log off come into my office please.

? q<CR>
Saved 1 message in /usr/tommy/mbox
```

If you try to execute **bug**, the program will execute up until the **mail $2** instruction. The **mail $2** instruction will then cause the **mail** command to execute interactively, as indicated by the **Subject:** prompt. However, the intention of **bug** is to send the mail message *automatically.*

To debug this program, try **sh** **–v**, which will print the lines of the file as they are read by the system.

```
$ sh -v bug tom tommy<CR>
today=`date`
person=$1
mail $2
Subject:
```

Notice that the output stops on the **mail** command. There is a problem with **mail**. The here document must be used to redirect input into **mail**.

Before you fix the **bug** program, try **sh -x**, which prints the commands and their arguments as they are read by the system.

```
$ sh -x bug tom tommy<CR>
+date
today=  Wed Jul 17 11:25:59 PDT 1985
person=tom
+ mail tommy
Subject:fR
```

Once again, the program stops at the **mail** command. Notice that the substitutions for the variables have been made and are displayed.

The corrected **bug** program is as follows:

```
$ cat bug<CR>
today=`date`
person=$1
mail $2 <<!
$person
When you log off come into my office please.
$today
MLH
!
$
```

The **tee** command is a helpful command to debug pipe lines. It places a copy of the output of a command into a file that you name, as well as piping it to another command. The general form of the **tee** command is:

command1 ¦ **tee save.file** ¦ *command2*<CR

save.file is the name of the file that will save the output of **command1** for you to study.

If you wanted to check on the output of the **grep** command in the following command line

who ¦ grep $1 ¦ cut –c1-9<CR>

you can use **tee** to copy the output of **grep** into a file to check after the program is done executing.

who ¦ grep $1 ¦ tee check ¦ cut –c1-9<CR>

The file **check** contains a copy of the output from the **grep** command.

```
$ who ¦ grep mlhmo ¦ tee check ¦ cut –c1-9<CR>
$ mlhmo
$ cat check<CR>
mlhmo    tty61    Apr 10    11:30
$
```

If you do a lot of shell programming, you will want to refer to the **ROS Programmer's Guide** and learn about command return codes and redirecting standard error.

MODIFYING YOUR SHELL ENVIRONMENT

The .profile File

When you log into the Bourne shell, a file in your login directory, named **.profile** is executed. The **.profile** is a Bourne shell program that issues commands to control your shell environment. The C-shell has a similar file, named **.login**. For information on the C-shell's **.login** file and other C-shell "dot" files, see the *csh*(1) pages of the *ROS Reference Manual* and the *C-Shell* section in the *ROS Programmer's Guide.*

Since the **.profile** is a file, it can be edited and changed to suit your needs. On some systems you can edit this file yourself, and on other systems the system administrator will do this for you.

If you can edit the file yourself, you may want to be cautious the first few times and make a copy of your **.profile** in another file called **safe.profile**.

```
$ cp .profile safe.profile<CR>
$
```

You can add commands to your **.profile** just as you can add commands to any other shell program. You can also set some terminal options with the **stty** command, and you can set some shell variables.

Adding Commands to .profile

How do you add commands to your **.profile**? Try this pleasant example. The ROS system will allow you to start out your day with a message from your computer. Edit your **.profile** and add the following **echo** command to the last line of the file.

Type in:

echo Good Morning!

Write and quit the editor.

Whenever you make changes to your **.profile** and you want to initiate them in the current work session, you may type in a . and space before **.profile**. The shell will reinitialize your environment, that is, it will read and execute the commands in your **.profile**.

Now, type in: **. .profile<CR>**

The system should respond with:

```
Good Morning!
$
```

Setting Terminal Options

The **stty** command can make your shell environment more convenient for you. The following are only two of the options available for **stty**.

stty tabs

This option prevents tabs from being expanded on output. This substantially reduces the number of characters output on terminals that understand tabs. Read the *stty*(1) pages in the *ROS Reference Manual* for more details.

stty echoi

This option causes your teminal to immediately echo characters from the type-ahead buffer to the screen.

If you want to use these options for the **stty** command, you create those command lines in your **.profile** just as you would create them in one of your the shell programs. If you use the **tail** command, which displays the last few lines of a file, you can see the results of adding those three command lines to your **.profile**.

```
$ tail -3 .profile<CR>
echo Good Morning!
stty abs
stty echoi
$
```

If you have not used the **tail** command before, the following is a brief recap of **tail**.

Command Recap

tail - display the last portion of a file

command	*options*	*arguments*
tail	*-n*	**file name**
Description:	Displays the last lines of a file.	
Options:	Using the option you can specify number of lines **n**. The default (no options) is ten lines. There are other options, besides specifying **-n**. You can specify blocks (b) or characters (c) instead of lines.	

Using Shell Variables

Several of the variables reserved by the shell are used in your **.profile**.

Let's take a quick look at four of these variables.

HOME

This variable gives the path for your login directory. Go to your login directory and type in **pwd<CR>**. What was the system response? Now type in **echo $HOME<CR>**. Was the system response the same as the response to **pwd**? **$HOME** is the default option for **cd**. If you do not specify a directory for **cd**, it will move you to **$HOME**.

PATH

This variable gives the system the search path for finding and executing commands.

If you want to see the current values for your **PATH** variable type in: **echo $PATH**.

```
$ echo $PATH<CR>
:/mylogin/bin:/bin:/usr/bin:/usr/lib
$
```

The **:** is a delimiter. Notice that for this **PATH** the system looks in **/mylogin/bin**, for the command first, then into **/bin**, then into **/usr/bin**, and so on.

If you are working on a project with several other people, you may want to set up a group **bin**, a directory of special shell programs used only by your group. The directory would be found from the root directory. The path would be **/group/bin**. How do you add this to your **PATH** variable? Edit your **.profile**, and add **:/group/bin** to the end of your **PATH**.

PATH=:/mylogin/bin:/bin:/usr/lib:/group/bin<CR>

TERM

This variable tells the shell what kind of terminal you are working on. If you have done any editing with **vi** you know that you have to specify:

TERM=*code***<CR>**

where *code* is your terminal type, such as **ridge**, **tvi950**, etc.

If you do not want to specify the **TERM** variable each time you log in, you can add those two command lines to your **.profile** and they will automatically be recognized each time you log into the ROS system. Or, if you log in on more than one type of terminal, you will want to add your **set.term** command to your **.profile**.

PS1

The **PS1** command allows you to change your prompt string. Try the following example. If you wish to use several words, remember to quote the phrase. Also, if you use quotes you can add a carriage return to your prompt.

Type in: **PS1="Enter command:<CR>"**

Now your prompt sign looks like:

```
$ . .profile<CR>
Enter command:
```

The mundane $ is gone forever, or until you delete the PS1 variable from your **.profile**.

CONCLUSION

This tutorial has given you the basics for creating some shell programs. If you have logged in and tried the examples and exercises as you read the tutorial, you can probably perform many of your day-to-day tasks with your new shell programs. Shell programming can be much more complex and perform more complicated tasks than shown in this brief tutorial. If you want to read further on shell commands and programming, read the *sh*(1) and *csh*(1) pages in the *ROS Reference Manual* and the *Shell* and *C-Shell* sections of the *ROS Programmer's Guide.*

SHELL PROGRAMMING EXERCISES

2-1. Make the command line

banner `date | cut –c12-16`<CR>

into a shell program called **time**.

2-2. Make a shell program that will give only the date in a banner display. Be careful what you name the program!

2-3. Make a shell program that will send a note to several people on your system.

2-4. Redirect the date command without the time into a file.

2-5. Echo the phrase "Dear colleague" in the same file as the date command without erasing the date.

2-6. Using the above exercises, make a shell program that will send a memo with:

- Current date and the "Dear colleague" at the top of the memo,
- Body of a file that is the memo, and
- Closing statement

to the same people on your system as in Exercise 2-3.

2-7. How would you **read** variables into the **mv.file** program.

2-8. Use the **for** loop to move a list of files in the current directory to another directory.

How would you move all files to another directory?

Ingredients:

```
*
$*
mv $file newdirectory
```

2-9. How would you change the program **search**, to search through several files?

Hint:

```
for file
in $*
```

2-10. Give yourself a new prompt that includes a carriage return. (Hint " <CR>")

2-11. Check to see what $HOME, $TERM, and $PATH are set to in your environment.

ANSWERS TO EXERCISES

COMMAND LANGUAGE EXERCISES

1-1. The * at the beginning of a file name will refer to all files that end in that file name, including that file name.

```
$ls *t<CR>
cat   123t   new.t   t
$
```

1-2. **cat [0-9]*** would display the files:

1memo
100data
9
05name

echo * will list all the files in the current directory.

1-3. You can place **?** any place in a file name.

1-4. **ls [0-9]*** will list only those files that start with a number.

ls [a-m]* will list only those files that begin with letters "a" through "m".

1-5. If you placed the sequential command line in the background mode, the immediate system response was the PID for the job.

No, the **&** must be placed at the end of the command line.

1-6. The command line would be:

cd; pwd > junk; ls >> junk; ed trial<CR>

1-7. Change the **-c** option of the command line to read:

banner `date ¦ cut –c1-10`<CR>

SHELL PROGRAMMING EXERCISES

2-1.

```
$cat time<CR>
banner `date | cut -c12-16`
$
$chmod u+x time<CR>
$ time<CR>
banner display of the time 10:26
$
```

2-2.

```
$cat mydate<CR>
banner `date | cut -c1-10`
$
```

2-3.

```
$cat tofriends<CR>
echo "Type in the name of the file containing the note."
read note
mail janice marylou bryan < $note
$
```

Or, if you wanted to use parameters for the logins.

```
$cat tofriends<CR>
echo "Type in the name of the file containing the note."
read note
mail $* < $note
$
```

2-4. **date ¦ cut –c1-10 > file1<CR>**

2-5. **echo Dear colleague >> file1<CR>**

2-6.

```
$cat send.memo<CR>
date | cut -c1-10 > memo1
echo Dear colleague >> memo1
cat memo >> memo1
echo A memo from M. L. Kelly >> memo1
mail janice marylou bryan < memo1
$
```

2-7.

```
$cat mv.file<CR>
echo type in the directory path
read path
echo "type in file names, end with <^d>"
while
read file
   do
   for file
       in $file
     do
          mv $file $path/$file
      done
   done
echo all done
$
```

2-8.

```
$cat mv.file<CR>
echo Please type in directory path
read path
for file
    in $*
do
    mv $file $path/$file
done
$
```

The command line would then be:

```
$ mv.file *<CR>
$
```

2-9. See hint.

```
$ cat search<CR>
for file
  in $*
  do
    if grep $word $file >/dev/null
    then echo $word is in $file
    else echo $word is NOT in $file
    fi
  done
```

2-10. Type the following command line into your **.profile**

PS1="Hello<CR>" <CR>

2-11.

```
$ echo $HOME<CR>
```

```
$ echo $TERM<CR>
```

```
$ echo $PATH<CR>
```

Chapter 7: FLOPPY DISK DRIVE COMMANDS

Chapter 7
FLOPPY DISK DRIVE COMMANDS

INTRODUCTION

This section describes the commands used to access the floppy disk drive. The floppy disk drive uses the standard UCSD Pascal format which allows 77 files and 1.2 Mbytes of storage on each disk.

FLOPPY DISK COMMANDS

The commands used with the floppy disk drive are:

dir	List directory
cat	concatenate files
cmp	compare two files
copyfloppy	copy the contents of one floppy disk to another
rm	remove files
zero	format floppy disk

DISPLAYING CONTENTS OF FLOPPY DISK

To list the floppy file names, insert a floppy disk in the drive, label side up and slotted edge first. Use **dir** as follows:

```
$ dir<CR>
back-1    9-Jul-85  2 Sided, Double Density, # Blocks: 2448
GOODFILE       41 10-May-83   10   81 Datafile
BADFILE        48 10-May-83   51  255 Datafile
<Unused>         2349              99
2 files, 99 used blocks, 2349 unused blocks
$
```

Notice that the UCSD format displays filenames in upper case characters. Unlike the format used to access files on the hard disk, UCSD filenames are not case-specific; you can use either upper- or lower-case characters to access these files.

In the example above, BADFILE occupies 48 512-byte sectors, was created May 10, starts at disk sector 51, and occupies 255 bytes of its last (48th) sector.

LISTING CONTENTS OF FILES

The contents of a floppy disk file can be listed using the **cat** command, followed by the device name of the floppy disk and the name of the file, as follows:

```
$ cat /dev/fl/filename
```

For example, to list the contents of the file **testfile**, enter:

```
$ cat /dev/fl/testfile<CR>
This is the contents of testfile
$
```

COPYING FILES

To copy a file from the working directory onto the floppy disk, use the **cat** program. **Cat** simply copies a file from one place to another:

```
$ cat filename > /dev/fl/filename
```

This command instructs the system to copy *filename* to a file of the same name on the floppy disk device (specified by **/dev/fl**).

To copy a file from the floppy disk into the working directory, use **dir** to verify the name of the file on the disk, then use **cat**. For example, to copy **badfile** from the floppy disk into your working directory, enter:

```
$ dir<CR>
back-1    9-Jul-85  2 Sided, Double Density, # Blocks: 2448
GOODFILE       41 10-May-83   10   81 Datafile
BADFILE        48 10-May-83   51  255 Datafile
<Unused>         2349              99
2 files, 99 used blocks, 2349 unused blocks
cat /dev/fl/badfile > badfile<CR>
$
```

WARNING

You cannot use the **cp** command to copy files to or from the floppy disk.

COPYING FLOPPY DISKS

The **copyfloppy** command allows you to copy the entire contents of a floppy disk to another floppy disk. The **copyfloppy** command has two options: **-c** and **-f**.

The **-c** (**c**runch) option compacts the files toward the beginning of the disk being copied to. This option is useful when you need more storage space for larger files. Since files are stored contiguously, you can gain more storage space on the disk by crunching the files toward the beginning of the disk and leaving a large contiguous space at the end of the disk blank.

The **-f** option formats the destination disk before attempting to write the contents of the source disk to it.

FORMATTING FLOPPY DISKS

If the floppy disk has never been used, it must be formatted for use with the Ridge 32. Before you can format the disk, a tab must be placed over the write-disable notch, as described in Chapter 1. Insert the floppy in the disk drive and enter the **zero** command as follows:

`$ zero -f` *volumename*

When you format your floppy disk, you must assign it a *volumename*, which is any name of seven or less characters.

WARNING
Formatting the floppy disk will remove all current files from the disk.

REMOVING AND COMPARING FILES

The **rm** (remove file) and **cmp** (compare files) commands operate as described in the *rm*(1) and *cmp*(1) pages of the *ROS Reference Manual.* However, filenames must be specified using the following syntax:

`/dev/fl/`*filename*

to specify that the files are on the floppy disk. If the filename is not preceded by the floppy disk designation, the operation will occur in your current working directory on the hard disk.

For example, to remove the file **testfile** from the floppy disk, enter:

```
$ rm /dev/fl/testfile<CR>
$
```

Chapter 8: ADMINISTRATIVE DUTIES

Chapter 8
ADMINISTRATIVE DUTIES

INTRODUCTION

This chapter is for the person responsible for the day-to-day operations of your organization's Ridge 32 computer system. This person is referred to as the System Administrator.

The System Administrator's duties include:

- Periodic data backups
- Monitoring disk usage
- Adding new users and devices to the system
- Correcting system failures
- Routine hardware maintenance

If you are the only user of the Ridge computer, you will have to act as the System Administrator.

THE SYSTEM CONSOLE

Most administrative activities are performed using the system console. If you are using a standard RS-232 terminal as your system console, it should be connected to the port labeled **J1** on the back of the Ridge computer. If you are using a Ridge graphics display as your system console, it must be designated as **disp0** by setting the device ID to 5 on that terminal's graphics interface board located in the Ridge card cage. By convention, this board is connected to the port labeled **display 1**. (See the *Ridge Hardware Reference Manual* for more information.)

Only one port can serve the system console. If both an RS-232 terminal and a Ridge graphics display are set to work as the system console, the graphics terminal will be selected.

From the system console, the System Administrator can:

- Install new software
- Restore the system after a system crash
- Receive system error messages

If only one terminal is connected to your Ridge 32 computer, it should be configured as the system console.

THE ROOT ACCOUNT

Most administrative activities must be done from a special account, called the **root** account.

From the **root** account, you have unrestricted access to any file or directory in the system, regardless of the security codes. The root-user can modify the codes and user and group ID names of any file or directory in the system.

To become the **root** user, log in as **root**:

```
login: root<CR>
password:root_password<CR>

Welcome to the Ridge - This is ROS 3.3
#
```

Notice the # system prompt, which indicates you are in the root account.

When a new system is shipped, **root** has no password. Create your root password by using the **passwd** command described in *ESTABLISHING CONTACT WITH THE ROS SYSTEM* in Chapter 3. After you establish a password for your **root** account with the **passwd** command, DON'T FORGET IT.

SUPER-USER ACCESS

As the System Administrator, you should also have a normal user account (typically */directory/***adm**) in which you perform all of your non-administrative tasks. ROS allows you to change your access level from that of a normal user to that of the **root** user (super-user) without logging out of your regular account.

To change your access permission to that of **root** user, use the **su** (set user) command without specifying a new user name and enter the password used by your **root** account, as follows:

```
$ su<CR>
password:root_password<CR>
#
```

Once you gain super-user access, your user name (the user name of your regular account) will remain the same. For example, if **starship** becomes a super-user, his user name will still be **starship**.

MONITORING DISK SPACE

The system administrator should check disk usage regularly. To find out what percentage of the disk is full, use **df** as in the following example:

```
# df<CR>
dev        kbytes   used  avail capacity mounted at
/dev/disc 141244  65524  75720    46%   /
#
```

If the "capacity" field, which shows the actual percentage of disk space used, shows 80% or more, system performance can be improved by removing files from the disk. Typically, individual user text files do not cause a significant disk usage problem, but there may be huge unused directories or data files whose removal would substantially reduce disk usage. It is wise to back up a file before removing it — attempting to reduce disk usage can end in accidental removal of important files.

EVER-EXPANDING LOG FILES

Some system log files accumulate data through time, and must be controlled by the administrator to prevent an eventual disk space shortage.

- **/usr/adm/sulog** — an ever-expanding log file to which a record of **su** usage is appended, but only if the super-user creates the directory **/usr/adm**.
- **/usr/lib/cron/log** — an ever-expanding log file to which **cron** activity and error messages are appended.
- **/etc/wtmp** — an ever-expanding log file to which login information, system reboots, and time changes are appended. This will only occur if the super-user creates the file.
- **/usr/spool/uucp/SYSLOG** and **/usr/spool/uucp/LOGFILE** — ever-expanding log files to which **uucp** communications data are appended. See the UUCP section in the *ROS Utility Guide* for details.

Periodically remove each log file or move it to a file of another name. This maintenance procedure could be accomplished by the following, if performed weekly:

mv *logfile logfile.lastweek*

With this move, *logfile* will never exceed one week's worth of data, and *logfile.lastweek* will contain last week's data until it is overwritten the following week.

ALTERING SYSTEM CONFIGURATION

THE BOURNE SHELL /etc/profile FILE

/etc/profile is a Bourne shell file that is executed each time a Bourne shell user logs in. This file is maintained by the **root** user and cannot be modified by the individual user. The standard **/etc/profile** sets certain environment variables, runs the **mail** and **news** programs, and prints the message of the day.

The following is an example of a **/etc/profile** script:

```
# "@(#)profile   1.3 3/14/85"

trap "" 1 2 3
export TZ LOGNAME TERM
if disptype -s
then
    TERM=ridge
    case `disptype` in
    0)  TERM=$TERM-m0 ;;
    1)  TERM=$TERM-m1 ;;
    2)  TERM=$TERM-m2 ;;
    *)
    esac
else
    TERM=tvi950
fi
TZ=PST8PDT
case "$0" in
-sh | -rsh)
    trap : 1 2 3
    cat /etc/motd
    trap "" 1 2 3
    if mail -e
    then echo "you have mail"
    fi
    if [ $LOGNAME != root ]
    then
        news -n
    fi
    ;;
-su)
    :
    ;;
esac
trap 1 2 3
```

In the script above, the conditional expression beginning with the line `if disptype -s` determines whether the terminal the user is logging in on is a Ridge Display or some other terminal type. The terminal type is then established using **TERM**. This expression must be modified if other types of terminals are to be used. Read the comments in **/usr/lib/terminfo/terminfo.src** to learn which abbreviation to use for your terminal.

If you are not using a TeleVideo or a Ridge terminal, you must use the **tic** command to create the entry (see the *tic*(1) page in the *ROS Reference Manual*).

The line `TZ=PST8PDT` means that this is the Pacific Standard Time zone, eight hours behind Greenwich Mean Time, and that Pacific Daylight Saving Time is observed in season. If Daylight Saving Time is never observed in a locality, omit the string following the time difference. In the time line above, you would remove `PDT`, leaving only `TZ=PST8`.

Individual users can not modify the **/etc/profile** file. A file similar to the **/etc/profile** file, called the **.profile** file, is available to each user of the Bourne shell. (See the *The .profile File* section in Chapter 4 and the Chapter 6 tutorial for more information.)

ADDING AN RS-232 TERMINAL

This procedure applies to RS-232 terminals only; it does not apply to the Ridge graphics display, or other non-RS-232 terminals.

General

RS-232 terminals are connected to the Floppy Disk / Line Printer (FDLP) board via RS-232 cables and the ports labeled **J1 - J4** located on the back panel of the Ridge computer enclosure. By default, characters are sent and received as 8 bits with no parity bit generated on output or expected on input. All RS-232 communication is handled in full duplex.

RS-232 Connections

The following lists the active pins and their functions on the **J1 - J4** ports.

pin 2 Transmit Data.

pin 3 Receive Data.

pin 7 Signal Ground.

pin 8 Carrier Detect. On a non-modem port this pin will be ignored. On a modem port, this pin must be high to enable communications; if it goes low, the communication will be terminated.

pin 20 DTR (Data Terminal Ready). This pin will be high, except to hang up when it is set to low for .5 of a second.

Since there is no "standard" RS-232 configuration, you must make sure the pins on your RS-232 cable are located in the correct positions for your terminal. Look at the user manual for your RS-232 terminal and compare the pin settings described above with those expected by your terminal. The Ridge RS-232 ports are configured as DTE (Data Terminal Equipment). When connecting DTE devices, such as an RS-232 terminal, the position of pins 2 & 3 on one end of the cable must be swapped. When connecting Data Communications Equipment (DCE), such as modems, use a "straight-through" cable with the same pin numbers on both ends.

Once you have your terminal cable configured correctly, connect the terminal to any of the 4 ports labeled **J1 - J4**. (Note that if no Ridge graphics display has been designated as the system console, **J1** will serve as the system console port; otherwise it is just like any of the other **J** ports.)

Terminal Settings

Since there is such a wide range of terminals, step-by-step instructions for configuring your terminal cannot be presented in this guide. The best this guide can do is explain the RS-232 characteristics of the Ridge. You will have to use this information in conjunction with the documentation provided with your terminal to determine the correct configuration for your terminal.

As stated at the beginning of this section, by default, characters are sent and received as 8 bits with no parity bit generated on output or read on input and that terminal communication is handled in full-duplex. Therefore, the internal switches of your terminal should be set to NO PARITY and FULL DUPLEX.

You should set the terminal's speed to either 9600, 2400, 1200, 300, or 19200 baud. Terminals connected to the Ridge via a modem typically require a 2400, 1200 or 300 baud setting. (See *The gettydefs File* section for more information on how the Ridge establishes baud rates.)

The remaining switches should be set as instructed by the documentation provided with your terminal.

The inittab File

The **/etc/inittab** file associates terminals with RS-232 ports. For each RS-232 terminal connected to the Ridge, there must be a corresponding entry in the **inittab** file. Dial-out lines, such as those used for modems and printers, do not have entries in the **inittab** file.

List the **/etc/inittab** file as follows:

```
# cat /etc/inittab<CR>
UM00::respawn:/etc/getty /dev/tty0 9600
UM01::respawn:/etc/getty /dev/tty1 9600
UM02::respawn:/etc/getty /dev/tty2 9600
#
```

The following briefly describes the entry format for each line in the **inittab** file:

id:rstate:action:process

The **id** field identifies the entry; the **rstate** field is traditionally used for process scheduling, but is currently not recognized by ROS; the **action** field contains a keyword that specifies how the process defined in the **process** field is to be handled; and the **process** field executes the **getty** program with the specified **line** and **label** parameters. (The **getty** program is discussed in the next section.)

The third entry in the example **inittab** file shown here has an **id** of **UM02**, no entry in the **rstate** field, specifies the **respawn** keyword in the **action** field, and executes the **getty** program with a **line** of **/dev/tty2** and a **label** of **9600**.

Each **id** field entry should be numbered consecutively (i.e., **UM00, UM01, ...**) for each line in the **inittab** file. The **line** parameters (**tty0, tty1, ...**) in the **process** field should be numbered to correspond to the **J#** ports on the back-panel. The ports labeled **J1**, **J2**, **J3**, and **J4** correspond to **tty0**, **tty1**, **tty2**, and **tty3**, respectively.

For example, if you have connected a fourth terminal to the **J4** port, you might add the following line to the **inittab** file shown above:

```
UM03::respawn:/etc/getty /dev/tty3 9600
```

Read the *inittab*(4) page of the *ROS Reference Manual* for full details on the **/etc/inittab** file.

The gettydefs File

The **/etc/gettydefs** file contains a list of entries that determine the initial baud rate, login message, and tty settings for RS-232 devices. Entries in the **gettydefs** file are read by the **getty** program during the system startup sequence.

After connecting a new RS-232 terminal to the Ridge, list the **/etc/gettydefs** file by entering:

```
# cat /etc/gettydefs<CR>
9600#B9600 CLOCAL#B9600 SANE TAB3 CLOCAL#login: #2400

2400#B2400 CLOCAL#B2400 SANE TAB3 CLOCAL#login: #1200

1200#B1200 CLOCAL#B1200 SANE TAB3 CLOCAL#login: #300

300#B300 CLOCAL#B300 SANE TAB3 CLOCAL#login: #19200

19200#B19200 CLOCAL#B19200 SANE TAB3 CLOCAL#login: #9600

modem1200#B1200#B1200 SANE TAB3 HUPCL#login: #modem300

modem300#B300#B300 SANE TAB3 HUPCL#login: #modem1200
#
```

The meaning of these lines is described in detail on the *gettydefs*(4) page in the *ROS Reference Manual.* The following briefly describes the entry format:

label#initial-flags#final-flags#login-prompt#next-label

Recall the **/etc/inittab** file described in the previous section. Each entry in the **/etc/inittab** file executes the **getty** program with **line** and **label** parameters that indicate an RS-232 line and a specific entry in the **gettydefs** file. The entry in the **gettydefs** file is located when **getty** matches its **label** parameter with a **label** field at the beginning of an entry in the **gettydefs** file. (For example, all of the entries in the example **inittab** file use the **9600** label, which specifies the first line in the **gettydefs** file above.)

If the baud rate specified by the initial **gettydefs** entry is incorrect for a terminal, the user can direct **getty** to try another entry in the **gettydefs** file by pressing the BREAK key on the terminal keyboard. This next entry is located by matching the contents of its **label** field with that of the **next-label** field in the initial entry. The user should continue pressing BREAK until the correct **gettydefs** line is located and the contents of the **login-prompt** field is printed on the screen.

In most cases, the **/etc/gettydefs** file need not be modified. If you cannot find an entry in the **gettydefs** file to match your terminal's baud rate and tty settings, read the *gettydefs*(4) page for information on creating a new entry in this file.

Activating New Terminals

The system must be rebooted in order to activate the new terminal(s). Follow the reboot procedure described in the *REBOOTING THE SYSTEM* section in this chapter.

Character Case

In order to establish communication with ROS, your terminal must be set to generate lowercase letters. This may be as simple as releasing the "caps lock" key. An uppercase-only terminal can be used on the Ridge 32 if the user converts uppercase to lowercase by entering **stty -iuclc < /dev/tty?** (where **?** is the appropriate **tty** number) from a lowercase terminal or a Ridge display. Unfortunately, the functionality of the uppercase terminal is still reduced because now it is impossible to generate uppercase characters when they are truly needed.

ADDING A RIDGE MONOCHROME DISPLAY

A Ridge Monochrome Display terminal need not be configured in software, but you may wish to activate the mouse pointer and window management software automatically at login time. This can be accomplished by adding the appropriate instructions to the **.profile** or **.login** file of those users with Ridge displays. Installation and mouse startup instructions are shipped with the display.

ADDING OTHER DEVICES

Most peripheral devices, such as the tape drive and DR11 interface board, require a software device driver to be running on the system. To automatically run the appropriate device drivers, modify the **/ros/conf** file.

```
:1:/drivers/fdlp
:32:/drivers/tapedriver
```

See *conf*(4) in the *ROS Reference Manual* for details on the fields in **/ros/conf**.

The physical connection of a Ridge-supported peripheral device is explained in the instructions shipped with that device, or is to be performed by the Ridge Systems Engineer.

ADDING A USER

A new user is specified by adding an entry to the **/etc/passwd** file. Each entry in the **passwd** file contains up to 7 fields, separated by colons. Below is a sample **passwd** file, the bold entry is for a guest user.

```
root:ynsSOvNHjtT6U:0:0:0000-Admin(0000):/:
daemon:xx:1:1:0000-Admin(0000):/:
  .
  .
shqer:xx:69:8:0000-rje(0000):/usr/rje:
trouble:xx:70:2:trouble(0000):/usr/lib/trouble:
lp:xx:71:2:0000-lp(0000):/usr/spool/lp:
ftp:xx:99:1:0000-ftp(0000):/usr/ftp:
starship:MJ0i0j/xrx/FY:100:100:Grace Jefferson:/user1/starship:
joe:IH4COYhjxLMOo:101:100:Joe Webster:/usr/joe:/bin/csh
george:c2FcJAmRzXUqI:102:100:George Turner:/user1/george:
guest::103:100:Guest User:/usr/guest:/bin/csh
```

The colon-separated fields in the **/etc/passwd** file are:

Login Name	This field specifies the character string the user will type after the `login:` prompt to gain access to his account.
Encrypted Password	This specifies the password the user will use. When you create a new account, do not make an entry in this field. When the user of the new account logs in and creates a password using the **passwd** command, an encrypted form of that user's password will be entered in this field. To make an account inactive, enter **xx** in this field, and no one will be able to log in to that account.
User ID	This is the ID number for the user. The user ID number is used by ROS to determine which files belong to which users.

Group ID	This is the ID number for the group. Users with the same group ID can share access to particular files and directories. The members of each group are specified in the **/etc/group** file. (See *group*(4) for more information.)
Optional Field	This field can contain any information you wish to associate with the user. Typically, it contains the user's full name.
Home Directory	This field identifies what directory will be the user's working directory when he logs in.
Shell Program	If the user uses the C-shell, or any shell other than the Bourne shell, the file containing the shell program can be specified in this field. If no entry is made in this field, the user will use the Bourne shell (**/bin/sh**).

The entry for **guest** in the sample **passwd** file shows that the login name is **guest**; no encrypted password has been created; the user ID is **103**; the group ID is **100**; the optional field identifies the account as belonging to **Guest User**; the initial working directory is **/usr/guest/**; and **/bin/csh** specifies that **guest** is to use the C-shell.

These fields are also described on the *passwd*(4) page of the *ROS Reference Manual.*

Now the administrator should create the home directory (as listed in the second to last field of **/etc/passwd**) for the new user and set its ownership using the following command forms:

```
mkdir /user_directory/username
chown username /user_directory/username
chgrp group id /user_directory/username
```

For example, to create a home directory for **joe** in the **usr** directory and assign him a group ID of 100, enter:

```
# mkdir /usr/joe<CR>
# chown joe /usr/joe<CR>
# chgrp 100 /usr/joe<CR>
#
```

Immediately after modifying **/etc/passwd** and making the new directory, the new user will have access to the system. Use the **passwd** command to create and change a password from a user account. If you damage the password field of the **passwd** file with the editor, or if you destroy the **passwd** file itself, call your Ridge Systems Engineer for directions on accessing the system without passwords.

MESSAGE OF THE DAY

/etc/motd contains the message each Bourne shell user sees upon logging into ROS. To modify this message, the **root** user may edit this file.

BACKING UP AND ARCHIVING DATA

Regularly scheduled backups of system data may prevent catastrophic data loss due to negligence or system failure.

It is not necessary to store the entire Ridge Operating System on tape because it is contained on floppy disks kept at your site. If your application software comes from Ridge, it will also be safe on floppy disk or tape at your site. User data and text files, and locally written program files must be backed up regularly.

Most system administrators do a complete backup of user directories every week. However, in some situations, it may be desirable to backup certain directories or files every day, or every few hours, depending on how often the data changes and how much of it you can afford to lose since the last backup.

ARCHIVE COMMANDS

The archive commands used when backing up files are:

- **cpio**
- **tar**

The cpio and tar Commands

Both the **tar** (tape archiver) and **cpio** (copy I/O) commands are used to archive and 'de-archive' files. **tar** is the original archive facility, which remains interchangeable with most UNIX systems. **cpio**, on the other hand, was developed later and, though easier to use, is not compatible with UNIX systems prior to System III.

This section will only discuss the **cpio** command. If you are interested in learning about the **tar** command, see the *tar*(1) pages in the *ROS Reference Manual.*

The **cpio** command has two forms:

cpio -o	Read a list of files from the standard input and write them, with pathname and status information, to standard output (the file representing the hard disk or tape device).
cpio -i *patterns*	Read files previously saved using **cpio -o** from the standard input. These files are specified using "patterns", which can be any combination of metacharacters mixed with alphabetic and numeric characters. These files are written into the current directory tree.

See the *cpio*(1) pages in the *ROS Reference Manual* for more information. Examples on the use of the **cpio** command are given later in the *BACKUP COMMANDS* section.

TAPE DEVICE FILES

There are special files that identify the tape drive unit number, bytes-per-inch (bpi) density value, and specify whether or not to rewind the tape when finished accessing it. These are called **mt** files. There are two general types of **mt** files:

- rmt
- mt

The **mt** files specify non-raw mode. In non-raw mode, characters are buffered by the driver and every tape record is exactly 4096 bytes long. You can read any number of bytes, but the driver will read physical records of 4096 and transfer to you the number requested. Non-raw mode is rarely used.

The **rmt** files specify raw mode. In raw mode, whatever size you write is the number of bytes in the next physical record. When you read in raw mode the driver always reads exactly one physical record, returning to you either exactly as many bytes as you asked for (if that matches the record size), fewer bytes (if the record was smaller than your request), or exactly as many bytes as you asked for and an error indicating that there were more (if the record was larger than your request).

Unless you have more than one tape drive, your unit number will be 0. Therefore, you will only be concerned with the following files:

raw mode					non-raw mode				
minor no.	bytes /inch	re-wind	unit no.	file name	minor no.	bytes /inch	re-wind	unit no.	file name
5	1600	n	0	rmt0	4	1600	n	0	mt0
7	1600	y	0	rmt1	6	1600	y	0	mt1
9	3200	n	0	rmt16	8	3200	n	0	mt16
11	3200	y	0	rmt17	10	3200	y	0	mt17

Tape device files are located in the **/dev** directory. Therefore, these files are typically identified using the form:

/dev/filename

For example, tape drive 0, with no rewind on close, raw mode, at 1600 bpi is identified by:

```
/dev/rmt0
```

See the *mt*(7) pages in the *ROS Reference Manual* for more information.

BACKUP COMMANDS

This section describes the commands you use to backup data using tapes or floppy disks. Once you have executed these commands and have determined which ones you will be using on a regular basis, you may wish to create aliases (if using the C-shell) or write shell programs to simplify the task of entering these commands.

Tape Backup

Use the following command form to back up specified directories from your hard disk to tape:

`find` *directory* -print ¦ cpio -ovB > /dev/rmt16

where *directory* is the name of the initial directory from to begin backing up data. The above command will save the contents of the initial directory and the contents of every subdirectory "below" it in the file system.

For example, to backup all of the directories owned by **starship** enter:

```
# find /user1/starship -print ¦ cpio -ovB > /dev/rmt16<CR>
list of filenames will appear
#
```

This command would backup the **/user1/starship** directory and all of the directories below it in the file system (i.e., **/user1/starship/draft**, **/user1/starship/letters**, etc..). The names of all of the files copied to the tape will be listed by the system as they are backed up.

You can also backup individual files. To specify an individual file to be backed up on tape, enter the directory pathname and the name of the file for the *directory* argument. For example, to backup the file**/user1/starship/mbox** you would enter:

```
# find /user1/starship/mbox -print ¦ cpio -ovB>/dev/rmt16<CR>
/user1/starship/mbox
#
```

Floppy Disk Backup

Backing up directories to floppy disk is very similar to backing them up on tape. However, in most situations, you will require several formatted disks. Therefore, before attempting your first backup, you must format several floppy disks using the **zero -f** command described in Chapter 7.

To estimate the number of floppy disks you should format, you can use the **du -s** command to determine the total number of blocks used by all of the files you wish to backup.

Each floppy disk holds over 1 Mbyte (1 million bytes) of data. Since each block represents 1 Kbyte (1,024 bytes), each disk can hold over 1,000 blocks. Once you determine the total number of blocks used by your files, you can roughly estimate the number of floppies required to store these files by the following formula:

```
number of blocks
----------------  = number of floppy disks (round down)
     1000
```

For example, to determine the number of blocks used by the user **starship**, you would enter:

```
# du -s /user1/starship<CR>
14580    /user1/starship
#
```

The total number of blocks for all of the files below **/user1/starship** is 14580 blocks, which means that you should format 14 floppy disks.

After formatting the correct number of floppy disks, backup the files to floppy disks using the following command form:

find *directory* -print | cpio -ov > /dev/fl/*filename*

where *directory* is the initial directory from which to begin backing up (as described in the *Tape Backup* section above) and *filename* is the name you wish to give the data on the disk.

For example, to backup all of the directories from the current working directory and name the resulting file **backup12** enter:

```
# find . -print | cpio -ov > /dev/fl/backup12<CR>
list of filenames will appear
#
```

If your current working directory is **/usr**, then entering this command would backup the directories under **/usr**.

When one disk is full, the system will prompt you to insert the next one and press the RETURN key. Should it be necessary to restore your backup directories after a hard disk failure or similar disaster, you must read the information from the floppy disks in the same order in which they were written to during the backup procedure. Because of this, it is important that you label each disk to indicate the order in which they are inserted during backup.

Compressed Data Backup

You may wish to make better use of the space on your floppy disk by compacting the data during backup. This involves sending the backup data to a file, compressing the file using the **compact** command and sending the compressed file to the floppy disk. This command sequence uses the form:

```
find directory -print | cpio -vo | compact > /dev/fl/filename
```

For example, if the directory you wish to backup is named **/user1/starship** and you want the backup file to be called **backup**, you would enter:

```
# find /user1/starship -print ¦ cpio -vo ¦ compact > /dev/fl/backup<CR>
list of filenames will appear
#
```

RESTORE COMMANDS

This section describes the commands used to transfer backed up data from tape or floppy disks to your hard disk.

Restoring From Tape

To restore backed up directories from tape, enter:

```
# cpio -imdBv < /dev/rmt16<CR>
list of filenames will appear
#
```

You can restore specific files from the tape using the form:

```
cpio -imdBv filename < /dev/rmt16
```

For example, if, after backing up the entire **/user1/starship** directory, you wish to restore the file **/user1/starship/file1**, you would enter:

```
# cpio -imdBv /user1/starship/file1 < /dev/rmt16<CR>
/user1/starship/file1
#
```

Restoring From Floppy Disks

To restore backed up directories from floppy disks, insert the first floppy disk in the backup series and enter:

```
# cpio -imdv < /dev/fl/filename<CR>
list of filenames will appear
#
```

where *filename* is the name given to the backup file. When the system is finished reading data from the first floppy disk, it will prompt you to insert the second floppy disk and continue.

You can restore a single file from a backup file stored on floppy disks using the form:

```
cpio -imdv single_filename < /dev/fl/backup_filename
```

For example, if, after backing up the entire **/user1/starship** directory under the filename **backup**, you wish to restore the file **/user1/starship/file1**, you would insert the first in the series of backup disks into the floppy disk drive and enter:

```
# cpio -imdv /user1/starship/file1 < /dev/fl/backup<CR>
/user1/starship/file1
#
```

If the file is not found on the first disk, the system will prompt you to insert the next disk. The system will continue to prompt you to insert the next disk in the backup series until the file is found and restored to the hard disk.

Restoring Compressed Data From Floppy Disk

To restore from a floppy disk in which the data was compacted, use the following command form:

```
uncompact /dev/fl/filename ¦ cpio -idvm
```

For example, to restore the **/user1/starship** directory that was backed up in the example for compressed data backup, enter:

```
# uncompact /dev/fl/backup ¦ cpio -idvm<CR>
list of filenames will appear
#
```

SYSTEM FAILURES

It may be difficult to determine between a system crash and a terminal failure. If one terminal indicates a failure and another one works, the Ridge processor has not failed. If this occurs, check to see if all of the setting are correct for that terminal by entering:

```
# stty -a < /dev/tty#<CR>
#
```

(where *tty#* is the stty file assigned to your terminal in the **inittab** file.) This command will display all of the current setting for the specified terminal.

If all of the tty processes seem to be running correctly, the terminal or terminal i/o hardware may be faulty.

If all of the terminals are inoperative, the system has crashed. If you are uncertain about what caused the system crash, you should contact the Ridge Customer Service department and describe your problem to a Ridge Systems Engineer (SE).

REGISTER DUMP PROCEDURE

After a complete system failure, the SE may ask you to perform the following steps:

1. Press the switch 0 (zero) button. This is the white button on the bottom of the clock board (see Figure 8-1), which is the half-sized circuit board at the far right end of the card cage. If your computer is equipped with an interface shield, this button is accessible through the right-most opening.
2. Go to the system console and notice the : prompt on the screen. This prompt indicates you have entered the RBUG debugger program.
3. The current value of the Program Counter (PC) will appear on the system console's display. Write this value down.
4. Enter **DR** to dump the user process registers. This will display the values of 16 general registers. Write down the sixteen 8-character numbers that appear on the screen.
5. Enter **DSR** to dump the kernel registers. This will display the current values of the code segment, data segment, and process control block. Write these values down.
6. Replace the front panel.
7. Press the blue LOAD button on the front panel. The system will then reboot.

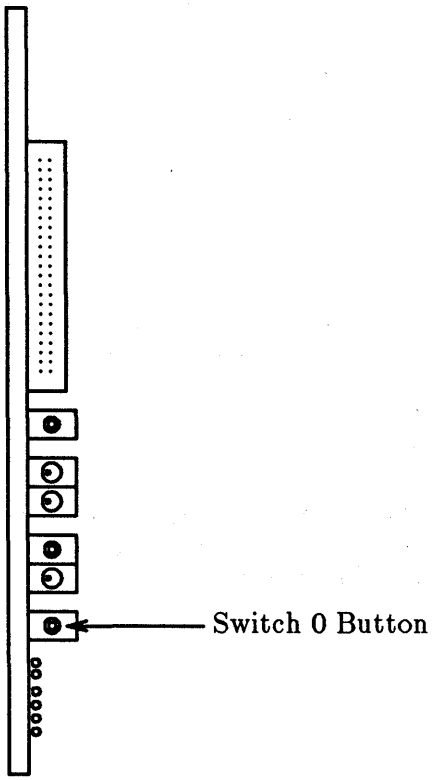


Figure 8-1. Location of Switch 0 Button on the Clock Board

REBOOTING THE SYSTEM

To reboot your computer, do the following:

1. Open the front panel of the Ridge 32, but do not open the metalic radio interference shield if so equipped.
2. Press the Switch 0 button on the clock board, as described in *REGISTER DUMP PROCEDURE*, Step 1.
3. Press the blue LOAD button on the Ridge 32 front panel,
4. After a while the system console will display a date followed by the prompt: `Do you want to run with this date? (y)`. If this date is correct, press the RETURN key. If you wish to set a new date, follow the procedure described in the *TURNING THE SYSTEM ON* section in Chapter 1.

After approximately 5 minutes, the `login:` prompt should reappear, indicating the system has been successfully rebooted.

ERROR MESSAGES

Most error messages displayed on your system console require the assistance of a Ridge Service Engineer. However, if any of the following hard disk I/O error messages is repeatedly displayed on the system console, you should certify the integrity of the hard disk.

```
I/O Error #7 Missing address mark
I/O Error #8 Can't find header that matches
I/O Error #9 CRC error in header
I/O Error #10 Uncorrectable error in data
I/O Error #11 Seek failure
```

Certify the integrity of the hard disk by the following procedure:

1. Back up the disk as described earlier in this chapter.
2. Obtain the SUS floppy disk and insert it in the floppy disk drive. For information on SUS operation, see the *SYSTEM DIAGNOSTICS* Chapter in the *Hardware Reference Manual.*
3. Ask all users to log off the system, and verify that they have done so.
4. On the Ridge 32 front panel, press and hold the DEV2 button, then press and release the LOAD button. Release DEV2 only after the red light on the floppy disk drive goes on.
5. At the system console, enter: **discutil**.
6. If the hard disk is a 78-Mb, 445-Mb, or other SMD-type drive, remove the SUS floppy and insert the "Bad Block" floppy that was shipped with the hard disk, and then enter: **RBB**. If the hard disk is a 60-Mb, 142-Mb, or other HD-type drive, just enter: **MBB**.
7. Now enter: **certify**

8. The **certify** utility may take 1 to 2 hours to complete. After **certify** completes, enter **LSP** to print a list of any bad spots that may have been detected during the check. Write the list down on paper. If no bad pages are reported, press the LOAD button on the Ridge 32 front panel.

If bad pages were reported, re-run the "certify" utility to verify the bad spots:

a. For each bad block on your hand-written list, enter

blocknumber-100, 200

For example, if **certify** reports a bad page at 37025, enter: **36925, 200**.

b. If **certify** reports a bad page at the same location again, this is a bona-fide bad block which must be entered on the bad blocks file on the disk (and on the floppy disk if it's a 445-Mb hard disk). In this case, call the Ridge Systems Engineer.

c. Press the LOAD button on the Ridge 32 front panel.

WARNING

NEVER stop **certify** prior to normal compilation. Doing so may cause one page of disk data to be destroyed. You can safely stop **certify** by toggling the load enable switch (second switch from the bottom on the clock board) from left to right and then back to the left.

SYSTEM UPGRADES, SOFTWARE UPDATES, AND SERVICE CONTRACTS

In some cases, new software or hardware products from Ridge Computers may be installed by the customer according to instructions supplied by the factory. However, many installation and upgrade duties are handled by a Ridge Systems Engineer if the customer has purchased a service contract.

Ridge Computers' service contract gives maximum hardware and licensed software support. Among other features, a service contract entitles the customer to:

- Parts, material, and labor to maintain the covered equipment in peak operating condition
- Upgrades of hardware equipment, software releases, and user documentation as they become available
- Periodic software bulletins with technical information related to Ridge software products

A service contract makes Ridge Computers responsible for maintaining the system correctly, and is considered indispensable by many customers.

For full details on the Ridge service contract program, contact the Ridge Customer Service department.

Chapter 9: PREVENTIVE MAINTENANCE

Chapter 9
PREVENTIVE MAINTENANCE

INTRODUCTION

To keep your system service contract valid, you must

- Maintain the environment at your computer site in accordance with the **Ridge Computer System Site Planning Guide**.
- Perform routine but simple preventive maintenance tasks on your own, as described in this chapter.

Read and follow all preventive maintenance instructions that are shipped with each peripheral product.

REPLACING THE RIDGE 32 AIR FILTER

Your air filter should be cleaned or replaced every three to six months.

On a Ridge 32/100:

1. Turn off and unplug the system.
2. Open the front panel of the Ridge 32/100, then open the radio interference shield with a phillips screw driver.
3. Remove the metallic air filter, clean it under running water, wait for it to dry, and replace it in the Ridge 32/100.
4. Close the radio interference shield and secure the phillips screws, then close and lock the front door of the Ridge 32/100.

On a Ridge 32/300:

1. Customers with a service contract may obtain an air filter free from the Ridge SE. The filter is an unusual size that you probably won't find at a hardware store.
2. Remove the six screws on the air filter cover on the back panel of the Ridge 32.
3. Remove and discard the dirty air filter.
4. Install a clean filter and replace the screws.

CLEANING THE EXTERIOR

As needed, clean the exterior of the Ridge 32 and peripheral equipment:

- Turn off and unplug the equipment.
- Use water, and/or a mild soap, and a non-abrasive cloth to remove fingerprints and food spills.

CLEANING THE READ/WRITE HEADS ON THE FLOPPY DISK DRIVE

1. Obtain the SUS floppy disk (the one you created according to the instructions under "Administrative Duties") and insert it in the floppy disk drive.
2. Ask all users to log off the system, and verify that they have done so.
3. On the Ridge 32 front panel, press and hold the DEV2 button, then press and release the LOAD button. Release DEV2 only after the red light on the floppy disk drive goes on.
4. Obtain a cleaning solution, such as isopropyl alchohol, and a floppy disk cleaner, such as a 8-inch cleaning disk.
5. Read the instructions on the cleaning disk, apply the solution to the cleaning disk and insert it in the drive.
6. At the SUS prompt, enter: `sh PMFD`
7. Verify the cleaning disk is in the slot and press the RETURN key.
8. Let the cleaning disk rotate in the disk drive for 30 seconds, then press LOAD on the front panel.
7. Remove the cleaning disk.

OTHER EQUIPMENT

If you have other equipment, such as a tape drive or 445-Mbyte disk in the Companion Enclosure, follow the maintenance instructions that come with that equipment.

GLOSSARY

This glossary defines terms and acronyms used in the *Ridge 32 User's Guide* that may not be familiar to you.

address

Generally, a number that indicates the location of information in the computer's memory. In the ROS system, the address is part of an editor command that specifies a line number or range.

append mode

A text editing mode where you enter (append) text after the current position in the buffer. See **text input mode**, compare with **command mode** and **insert mode**.

argument

Special instructions on the command line that specify data on which a command is to operate. Arguments usually follow the command and can include numbers, letters, or text strings. For instance, in the command **lp −m myfile**, **lp** is the command and **myfile** is the argument. See **option**.

ASCII

(pronounced **as'-kee**) American Standard Code for Information Interchange, a standard for data transmission that is used in the ROS system. ASCII assigns sets of 0s and 1s to represent 128 characters, including alphabetical characters, numerals, and standard special characters, such as #, $, %, and &.

background

A type of program execution where you request the shell to run a command away from the interaction between you and the computer ("in the background"). While this command runs, the shell prompts you to enter other commands through the terminal.

baud rate

A measure of the speed of data transfer from a computer to a peripheral device (such as a terminal) or from one device to another. Common baud rates are 300, 1200, 4800, and 9600. As a general guide, divide a baud rate by 10 to get the approximate number of English characters transmitted each second.

Bourne shell

The original UNIX shell developed by AT&T Bell Laboratories. ROS uses this shell by default.

buffer

A temporary storage area of the computer used by text editors to make changes to a copy of an existing file. When you edit a file, its contents are read into a buffer, where you make changes to the text. For the changes to become a part of the permanent file, you must write the buffer contents back into the file. See **permanent file**.

C-shell
A shell developed at University of California, Berkeley. The C-shell interface is similar to programming in the C language and is, therefore, preferred by some programmers.

child directory
See **subdirectory**.

command
The name of a file that contains a program that can be processed or executed by the computer on request.

command file
See **executable file.**

command language interpreter
A program that acts as a direct interface between you and the computer. In the ROS system, a program called the **shell** takes commands and translates them into a language understood by the computer.

command line
A line containing one or more commands, ended by typing a carriage return (<**CR**>). The line may also contain options and arguments. You type this line to the shell to instruct the computer to perform one or more tasks.

command mode
A text editing mode in which each character you type is interpreted as an editing command. This mode permits actions such as moving around in the buffer, deleting text, or moving lines of text. See **text input mode**, compare with **append mode** and **insert mode.**

context search
A technique for locating a specified pattern of characters (called a string) when in a text editor. Editing commands that cause a context search scan the buffer, looking for a match with the string specified in the command. See **string.**

control character
A nonprinting character that is entered by holding down the control key and typing a character. A control character transmits a special command to the computer. For instance, when viewing a long file on your screen with the **cat** command, typing control-s (^**s**) stops the display so you can read it, and typing control-q (^**q**) continues the display.

current directory
The directory in which you are presently working. You have direct access to all files and subdirectories contained in your current directory. The shorthand notation for the current directory is a dot (.).

cursor
A cue printed on the terminal screen that indicates the position at which you enter or delete a character. It is usually a rectangle or a blinking line.

default

An automatically assigned value or condition that exists unless you explicitly change it. For example, the shell prompt string has a default value of $ unless you change it.

delimiter

A character that logically separates items or arguments on a command line. Two frequently used delimiters in the ROS system are the space and the tab. Another is the slash character (/) that separates directories from subdirectories and files in a path name.

diagnostic

A message printed at your terminal to indicate an error encountered while trying to execute some command or program. Generally, you need not respond directly to a diagnostic message.

directory

A type of file used to group and organize other files or directories. You cannot enter text or other data into a directory. (For more detail, see Chapter 4.)

disk

A magnetic data storage device consisting of several round plates similar to phonograph records. Disks store large amounts of data and allow quick access to any piece of data.

electronic mail

The feature of an operating system that allows computers users to exchange written messages via the computer. The ROS system **mail** command provides electronic mail in which the addresses are the login names of users.

environment

The conditions under which you work while using the ROS system. Your environment includes those things that personalize your login and allow you to interact in specific ways with the ROS system and the computer. For example, your shell environment includes such things as your shell prompt string, specifics for backspace and erase characters, and commands for sending output from your terminal to the computer.

erase character

The character you type to delete the previous character on the current line. The ROS system default erase character is #.

escape

A means of getting into the shell from within a text editor or another program.

execute

The computer's action of interpreting a programmed instruction or command and performing the indicated operation(s).

executable file

A file that can be processed or executed by the computer without any further translation. When you type in the file name, the commands in the file are executed. See **shell procedure**.

field

A word or a group of characters treated as one word on a command line. Fields are usually a fixed number of character positions in size, but they may also vary.

file

A collection of information. Files may contain data, programs, or other text. You access ROS system files by name. See **ordinary file**, **permanent file**, and **executable file**.

file name

A sequence of characters that denotes a file. (In the ROS system, a slash character (/) cannot be used as part of a file name.)

file system

A collection of files and the structure that links them together. The file system is a hierarchical structure--that is, a ranked system of files. (For more detail, see Chapter 4.)

filter

A command that reads the standard input, acts on it in some way, and then prints the result as standard output.

final copy

The completed, printed version of a file of text.

foreground

The normal type of program execution. In foreground mode, the shell waits for a command to end before prompting you for another command. In other words, you enter something into the computer and the computer "replies" before you enter something else.

full-duplex

A type of data communication in which a computer system can transmit and receive data simultaneously. Terminals and modems usually have settings for half-duplex (one-way) and full-duplex communication; the ROS system uses the full-duplex setting.

full path name

A path name that originates at the root directory of the ROS system and leads to a specific file or directory. Each file and directory in the ROS system has a unique full path name, sometimes called an absolute path name. See **path name**.

global

A qualifier that indicates the complete or entire file. While normal editor commands commonly act on only the first instance of a pattern in the file, global commands perform the action on all instances in the file.

hardware

The physical machinery of a computer and any associated devices.

hidden character

One of a group of characters within the standard ASCII set, but not normally printed as visible symbols. Control characters, such as backspace and escape, are examples.

home directory
The directory in which you are located when you log in to the ROS system; also known as your login directory.

input/output
The path by which information enters a computer system (input) and leaves the system (output). An input device that you use is the keyboard and an output device is the terminal monitor.

insert mode
A text editing mode in which you enter (insert) text before the current position in the buffer. See **text input mode**, compare with **append mode** and **command mode**.

interactive
Describes an operating system (such as the ROS system) that can handle immediate-response communication between you and the computer. In other words, you interact with the computer from moment to moment.

line editor
An editing program in which text is operated upon on a line-by-line basis within a file. Commands for creating, changing, and removing text use line addresses to determine where in the file the changes are made. Changes can be viewed after they are made by displaying the lines changed. See **text editor**, compare with **screen editor**.

login
The procedure used to gain access to the ROS operating system.

login directory
See **home directory**.

login name
A string of characters used to identify a user. Your login name is different from other login names.

log off
The procedure used to exit from the ROS operating system.

metacharacter
One of a group of characters with a special meaning to the **shell**, such as < > * ? ¦ & $; ( ) \ " ` ' [ ] .

mode
In general, a particular type of operation (for example, an editor's append mode). In relation to the file system, a mode is an octal number used to determine who can have access to your files and what kind of access they can have. See **permissions**.

modem
A device that connects a terminal and a computer by way of a telephone line. A modem converts digital signals to tones and converts tones back to digital signals, allowing a terminal and a computer to exchange data over standard telephone lines.

multitasking
The ability of an operating system to execute more than one program at a time.

multiuser
The ability of an operating system to support several users on the system at the same time.

nroff
A text formatter available as an add-on to the ROS system. You can use the **nroff** program to produce a formatted on-line copy or a printed copy of a file. See **text formatter**.

operating system
The software system on a computer under which all other software runs. The ROS system is an operating system.

option
Special instructions that modify how a command runs. Options are a type of argument that follow a command and are preceded by a minus sign (–). You can specify more than one option for any command given in the ROS system. For example, in the command **ls –l –a directory**, **–l** and **–a** are options that modify the **ls** command. See **argument**.

ordinary file
A collection of one to several thousand characters. Ordinary files may contain text or other data but are not executable. See **executable file**.

output
Information processed in some fashion by a computer and delivered to you by way of a printer, a terminal, or a similar device.

parameter
Generally, a value that determines the characteristics or behavior of something. In the ROS system, a type of variable found only on the command line. See **variable**.

parent directory
The directory immediately above a subdirectory or file in the file system organization. The shorthand notation for the parent directory is two dots (..).

parity
A method used by a computer for checking that the data received matches the data sent.

password
A code word known only to you that is called for in the login process. The computer uses the password to verify that you may indeed use the system.

path name
A sequence of directory names separated by the slash character (/) and ending with the name of a file or directory. The path name defines the connection path between some directory and a file.

peripheral device
Auxiliary devices under the control of the main computer, used mostly for input, output, and storage functions. Some examples include terminals, printers, and disk drives.

permanent file

The data stored permanently in the file system structure. To change a permanent file, you must make use of a text editor, which maintains a temporary work space, or buffer, apart from the permanent files. Once changes have been made to the buffer, they must be written to the permanent file to make the changes permanent. See **buffer**.

permissions

Access modes, associated with directories and files, that permit or deny system users the ability to read, write, and/or execute the directories or files. You determine the permissions for your directories or files by changing the mode for each one with the **chmod** command.

pipe

A method of redirecting the output of one command to be the input of another command. It is named for the character (¦) that redirects the output. For example, the shell command **who ¦ wc –l** pipes output from the **who** command to the **wc** command, telling you the total number of people logged into your ROS system.

pipeline

A series of filters separated by the pipe character (¦). The output of each filter becomes the input of the next filter in the line. The last filter in the pipeline writes to its standard output. See **filter**.

positional parameters

Variables that hold arguments supplied with a shell procedure. They are placed into variable names, such as **$1**, **$2**, and **$3** when the shell calls for the shell procedure. The name of the shell procedure is positional parameter **$0**. See **variable** and **shell procedure**.

prompt

A cue displayed at your terminal by the shell, telling you that the shell is ready to accept your next request. The prompt can be a character or a series of characters. The ROS system default prompt is the dollar sign character (**$**).

printer

An output device that prints the data it receives from the computer on paper.

process

Generally a program that is at some stage of execution. In the ROS system, it also refers to the execution of a computer environment, including contents of memory, register values, name of the current directory, status of files, information recorded at login time, and various other items.

program

The instructions given to a computer on how to do a specific task. Programs are user-executable software.

read-ahead capability

The ability of the ROS system to read and interpret your input while sending output information to your terminal in response to previous input. The ROS system separates input from output and processes each correctly.

relative path name
The path name to a file or directory which varies in relation to the directory in which you are currently working.

remote system
A system other than the one on which you are working.

root
The source of all files and directories in the file system, designated by a slash character (/).

ROS system
A general-purpose, multiuser, interactive, time-sharing operating system based on the UNIX system developed by AT&T Bell Laboratories. The ROS system allows limited computer resources to be shared by several users and efficiently organizes the user's interface to a computer system.

screen editor
An editing program in which text is operated on relative to the position of the cursor on a visual display. Commands for entering, changing, and removing text involve moving the cursor to the area to be altered and performing the necessary operation. Changes are viewed on the terminal display as they are made. See **text editor**, compare with **line editor**.

search pattern
See **string**.

search string
See **string**.

secondary prompt
A cue displayed at your terminal by the shell to tell you that the command typed in response to the primary prompt is incomplete. The ROS system default secondary prompt is the "greater than" character (>).

shell
A ROS system program that handles the communication between you and the computer. The shell is also known as a command language interpreter because it translates your commands into a language understandable by the computer. The shell accepts commands and causes the appropriate program to be executed.

shell procedure
An executable file that is not a compiled program. A shell procedure calls the shell to read and execute commands contained in a file. This lets you store a sequence of commands in a file for repeated use. It is also called a command file. See **executable file**.

shell script
See **shell procedure**.

silent character
See **hidden character**.

software
Instructions and programs that tell the computer what to do. Contrast with **hardware**.

source code
The English-language version of a program. The source code must be translated to machine language by a program known as a compiler before the computer can execute the program.

special character
See **metacharacter**.

special file
A file (called a device driver) used as an interface to an input/output device, such as a user terminal, a disk drive, or a line printer.

standard input
An open file that is normally connected directly to the keyboard. Standard input to a command normally goes from the keyboard to this file and then into the shell. You can redirect the standard input to come from another file instead of from the keyboard; use an argument in the form < **file**. Input to the command will then come from the specified file.

standard output
An open file that is normally connected directly to a primary output device, such as a terminal printer or screen. Standard output from the computer normally goes to this file and then to the output device. You can redirect the standard output into another file instead of to the printer or screen; use an argument in the form > **file**. Output will then go to the specified file.

string
Designation for a particular group or pattern of characters, such as a word or phrase, that may contain special characters. In a text editor, a context search interprets the special characters and attempts to match a specified string with an identical string in the editor buffer.

string value
A specified group of characters that is symbolized to the shell by a variable. See **variable**.

subdirectory
A directory pointed to by a directory one level above it in the file system organization; also called a child directory.

system administrator
The person who monitors and controls the computer on which your ROS system runs; sometimes referred to as a super-user.

system console
The terminal from which the system is initiated, debugged, and monitored. This terminal is connected to either the port labeled **J1** (if a standard ASCII, RS-232 terminal) or **display 1** (if a bit-map graphics display terminal).

terminal
An input/output device connected to a computer system, usually consisting of a keyboard with a video display or a printer. A terminal allows you to give the computer instructions and to receive information in response.

text editor
Software for creating, changing, or removing text with the aid of a computer (known as text processing). Most text editors have two modes--an input mode for typing in text, and a command mode for moving or modifying text. Two examples are the ROS system editors **ed** and **vi**. See **line editor** and **screen editor**.

text formatter
A program that prepares a file of text for printed output. To make use of a text formatter, your file must also contain some special commands for structuring the final copy. These special commands tell the formatter to justify margins, start new paragraphs, set up lists and tables, place figures, and so on. Two text formatters available as add-ons to your ROS system are **nroff** and **troff**.

text input mode
A text editing mode where the text you type is added into the buffer. To execute a command, you must leave the input mode. See **command mode**, compare with **append mode** and insert mode.

timesharing
A method of operation in which several users share a common computer system seemingly simultaneously. The computer interacts with each user in sequence, but the high-speed operation makes it seem that the computer is giving each user its complete attention.

tool
A package of software programs.

troff
A text formatter available as an add-on to the ROS system. The **troff** program drives a phototypesetter to produce high-quality printed text from a file. See **text formatter**.

tty
Historically, the abbreviation for a teletype terminal. Today, it is generally used to denote a user terminal.

user
Anyone who uses a computer or an operating system.

user-defined
Something determined by the user.

user-defined variable
A shell name given by the user for the value of a string of characters. See **variable**.

utility
Software used to carry out routine functions or to assist a programmer or system user in establishing routine tasks.

variable

A symbol whose value may change within a program or a repetition of a program. In the shell, a variable is a name representing some string of characters (a **string value**). Some variables are normally set only on a command line and are called **parameters** (**positional parameters** and **keyword parameters**). Other variables are simply names to which the user (**user-defined variables**) or the shell itself may assign string values. (Keyword parameters are discussed fully in the *Shell* and *C-Shell* sections in the *ROS Programmer's Guide.*)

video display terminal

A terminal that uses a televisionlike screen (a monitor) to display information. A video display terminal can display information much faster than printing terminals.

visual editor

See **screen editor**.

working directory

See **current directory**.

INDEX

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y & Z

RIDGE

Reader Feedback

Manual Title: _____

Overall rating of this document	☐ Excellent	☐ Adequate	☐ Poor
Readability of text	☐ Very Clear	☐ Adequate	☐ Difficult
Usefulness of information	☐ Helpful	☐ Adequate	☐ Not Useful
Any errors?	☐ Yes	☐ No	

If so, identify error and page number _____

Additional comments: _____

Would you like more information on Ridge products? _____

Name _____

Company _____

Address _____

City _____ State _____ Zip _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1590 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Ridge Computers

Publications Department
2451 Mission College Blvd.
Santa Clara, CA 95050-8290

Ridge Computers
2451 Mission College Blvd.
Santa Clara, CA 95054
(408) 986-8500