Forty Dollars

Cromemco®

Introduction to Cromix-Plus

Reference Manual

Cromemco®

Introduction to Cromix-Plus

Reference Manual

023-5012

October 1987

Rev. F

CROMEMCO, Inc.
P.O. Box 7400
280 Bernardo Avenue
Mountain View, CA 94039

Copyright © 1986
CROMEMCO, Inc.
All Rights Reserved

This manual was produced using a Cromemco System 420 computer running under the Cromemco UNIX Operating System. The text was edited with the Cromemco CE Editor. The edited text was formatted by the UNIX TROFF formatter and printed on a Texas Instruments OmniLaser 2108 Printer.

The following are registered trademarks of Cromemco, Inc.

C-Net®
Cromemco®
Cromix®
FontMaster®
SlideMaster®
SpellMaster®
System Zero®
System Two®
System Three®
WriteMaster®

The following are trademarks of Cromemco, Inc.

C-10™
CalcMaster™
Cromix-Plus™
DiskMaster™
Maximizer™
TeleMaster™
System One™
System 100™
System 120™
System 200™
System 220™
System 400™
System 420™

UNIX is a registered trademark of Bell Laboratories.

CONTENTS

Chapter 1 - Introduction 1

Chapter 2 - Getting Started 1
2.1 Before You Begin 1
2.2 The Terminal Keyboard 1
2.3 Logging in to the Cromix-Plus System 2
2.4 Some Conventions Used in this Manual 3
2.5 Giving Some Typical Commands 4
2.6 Command Arguments 5
2.7 Command Syntax 5
2.8 Displaying the On-line Manual 5
2.9 Stopping a Program While it is Executing 7
2.10 Displaying Additional Screens of Information 7
2.11 The Terminal as a Carriage-Return Device 7
2.12 Error Messages 8
2.13 Changing Your Password 8
2.14 Logging Out 9
2.15 Intrinsic Commands 10
2.16 Privileged Access 10

Chapter 3 - Working With Text Files 1
3.1 Naming Files 1
3.2 An Additional Consideration When Naming Files 2
3.3 Special Characters 2
3.4 Creating a Sample File 2
3.5 Displaying a List of Files 3
3.6 Command Options 3
3.7 Access Privileges for the Owner of a File 6
3.8 Displaying the Contents of a File 7
3.9 Displaying a File with Page Headings and Line Numbers 7
3.10 Making a Copy of a File 8
3.11 File Links 9
3.12 Renaming a File 10
3.13 Deleting a File 10
3.14 Some Common Error Messages 11
3.15 Printing a File 12
3.16 General Rules for Command Options 14
3.17 The .bin, .com, and .cmd Filename Extensions 15

Chapter 4 - Cromix File Structure 1
4.1 The Home Directory 1
4.2 Visualizing the Cromix-Plus File Structure 2
4.3 Absolute Pathnames 5
4.4 How to Make Sure You Have Execute Access for a Directory . . . 8
4.5 Displaying the Absolute Pathname of an Executable File 9
4.6 Relative Pathnames 10

4.7 Changing Directories . . . 11
4.8 Creating a Directory . . . 13
4.9 Moving Files to a Directory . . . 14
4.10 How Move Works . . . 14
4.11 Copying Files to Another Directory . . . 14
4.12 Renaming Files With Move and Copy . . . 15
4.13 Shortcuts for Working Within a Directory Structure . . . 17
4.14 Deleting a Directory Structure . . . 19
4.15 Copying a Directory Structure . . . 20
4.16 How the Shell Looks for Executable Files . . . 22
4.17 Special Files in the Home Directory . . . 23
4.18 Device Files . . . 25

Chapter 5 - The Mail Utility . . . 1
5.1 Sending Mail . . . 1
5.2 Correcting Mistakes While Using Mail . . . 2
5.3 What Happens to the Mail You Send . . . 2
5.4 How Do You Know When You Have Mail . . . 3
5.5 Reading Your Mail . . . 3
5.6 The mbox File . . . 4
5.7 Sending the Same Mail to Several Users . . . 4

Chapter 6 - The Cromix-Plus Shell . . . 1
6.1 The Standard Output . . . 2
6.2 The Standard Input . . . 2
6.3 The Sort Utility . . . 3
6.4 Redirecting Output to a File . . . 4
6.5 Appending Output to a File . . . 6
6.6 Redirecting Type's Output to a File . . . 7
6.7 Redirecting Input From a File . . . 8
6.8 Running a Job in the Background . . . 9
6.9 Giving Sequential Commands . . . 10
6.10 Parentheses on the Command Line . . . 11
6.11 Redirecting Error Messages . . . 11
6.12 Redirection with Pipes . . . 12
6.13 Redirecting Output to a Temporary File . . . 13
6.14 The Tee Command . . . 13
6.15 Filename Generation . . . 14
6.16 Specifying a Range of Characters . . . 15
6.17 An Important Consideration Regarding Filename Generation . . . 17
6.18 Experimenting with Filename Generation . . . 17

Chapter 7 - Writing Command Files . . . 1
7.1 Command-File Description . . . 1
7.2 A Practical Use of the Path Command . . . 2
7.3 Redirection Within a Command File . . . 3
7.4 The Echo Command . . . 3
7.5 Command File Structure . . . 4
7.6 The Goto Command . . . 7

7.7 The If Command 7
7.8 The Shift Command 9
7.9 The Rewind Command 9
7.10 The Exit Command 9
7.11 The Input and Strcmp Commands 10
7.12 The Strcmp Command 10
7.13 The Repeat Command 11
7.14 The Scan Command 12
7.15 Sample Command Files 13

Appendix A - The Shell Command-Line Editor 16
A.1 Retrieving the Previous Command 16

Chapter 1 - Introduction

The Cromemco Cromix-Plus Operating System is a program that supervises the operation of the computer and its resources, disk drives, printers, terminals, modems, and so on. All computers, even the personal computer you may have at home, require some kind of operating system.

The Cromix-Plus system stores information in files--each containing some related information, and each having a distinct filename. (Physically, these files are located on some storage medium, such as the computer's hard disk.) To ensure that files are easy to access, the Cromix-Plus system organizes all files in directories within a hierarchical file structure (chapter 4).

Password security and a system of access privileges protect files from unauthorized access. If desired, access privileges can be changed to authorize access by selected users.

At any given time, a personal computer can normally do only one job for one user. A Cromemco microcomputer, on the other hand, can be working on numerous jobs for many users. The Cromix-Plus system is a multiuser system. While the computer compiles a program for one user, it may be printing a file for another. From the viewpoint of the Cromix-Plus user, the computer is dedicated to his or her special needs.

Cromix-Plus is also a multitasking operating system. To illustrate, suppose you need to print 10 copies of a lengthy report, and you also need to edit an important file. By printing the report as a "background" task, you can free your terminal for other work, such as editing that file. This is "multitasking," and it means you won't waste time waiting for one job to finish so you can start another.

With some multi-user systems, as the number of users and tasks increase, it takes longer and longer to process individual jobs. Such degradation in performance can be a real problem. With the Cromix-Plus Operating System, degradation is minimal because the Cromix-Plus system takes full advantage of the Cromemco computer's 68000 microprocessor family.

Supplied with every Cromix-Plus system is a set of utility programs. These utilities handle the jobs users need done again and again, such as printing a copy of a file (the Spool utility) or giving a file a new filename (the Rename utility). The most versatile of Cromix-Plus utility programs is the Shell utility. The Shell provides the interface between the Cromix-Plus Operating System and its users. It is the Cromix-Plus command interpreter and command processor. The Shell is also programmable, which means you can instruct it to process commands in special ways. Such customized commands can be given from the command line (chapter 6) or from command files (chapter 7).

The Cromix-Plus Operating System is as powerful as many operating systems designed to run on large,

mainframe computers. Its capabilities include:

- Device-compatible I/O, to support redirection of input and output.
- Date and time support.
- Numerous file buffers for high-speed execution.
- Resident execution of tasks (jobs are not swapped out to disk).
- RAM-disk feature, providing extra RAM for use as a high-speed disk.

Chapter 2 - Getting Started

In this chapter, you'll start using the computer and give some simple commands to the operating system. In the process, you'll become familiar with your terminal keyboard and some of the conventions used in this manual.

If you've used a computer with a multi-user operating system before, much of the information in this chapter will seem familiar. Be sure to at least skim through the chapter before continuing.

2.1 Before You Begin

Talk to the System Administrator--the person in charge of your Cromix-Plus System. The System Administrator will create a user account and a directory for you.

The System Administrator will also help you choose a login name and password. Your login name identifies you to the operating system--your password ensures that only *you* can log in with your login name.

When you have received a login name and password, you can log in to the system. In other words, you can tell the Cromix-Plus system you wish to use the computer, and the system will grant your request.

2.2 The Terminal Keyboard

Figure 2-1, a Cromemco 3102 terminal keyboard, points out some important keys.

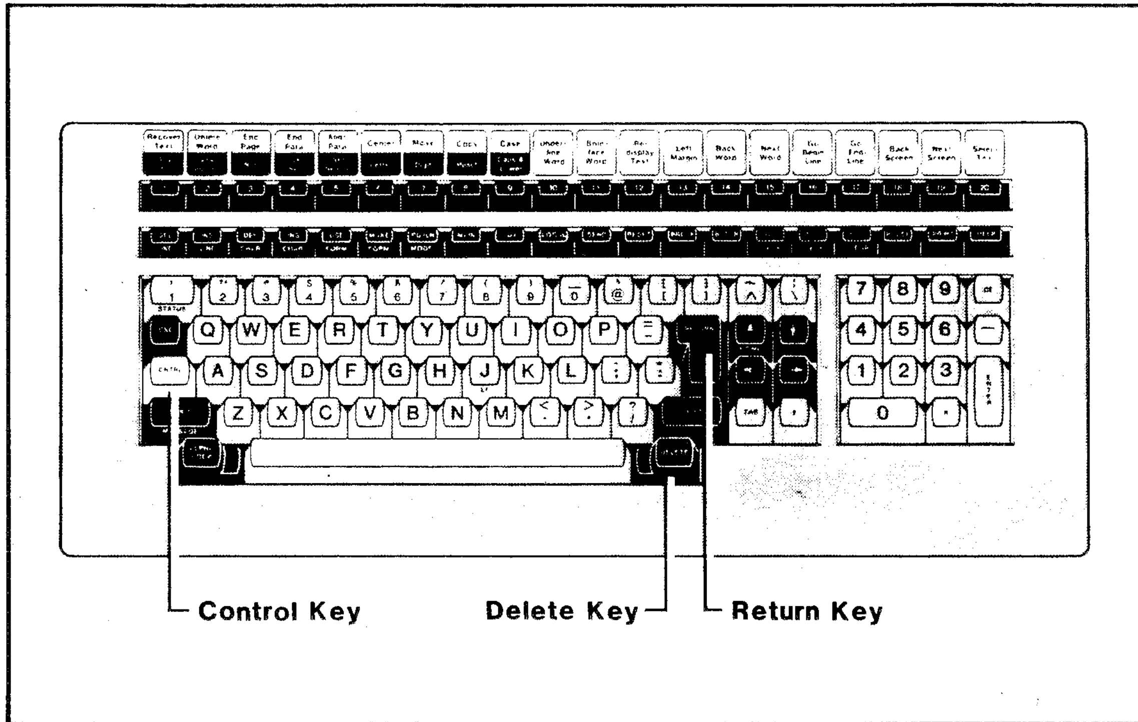

Figure 2-1: CROMEMCO 3102 TERMINAL

The RETURN key is an end-line key. After you've typed an instruction to the operating system, pressing RETURN ends the typed line and enters that instruction. As you're typing *text* from the keyboard, a RETURN ends the current line so you can begin a new line.

The DELETE key deletes characters on the current line. (DELETE is not an "abort," or program-interrupt, key as it is with some other operating systems.)

The CONTROL key is discussed later in the chapter.

2.3 Logging in to the Cromix-Plus System

Many terminals may be connected to a multi-user Cromix-Plus system, and the system keeps track of what terminals are in use. When a terminal is not in use, the system displays the prompt "Login:". This prompt means that you (or any system user) can log in to the Cromix-Plus system from that terminal.

You should see "Login:" on the terminal now. If you do not, make sure the terminal is turned on (there is an ON/OFF switch at the rear of the terminal). Then press the RETURN key on the terminal keyboard a few times.

With the cursor next to "Login:", type your login name. Because the Cromix-Plus system is not

sensitive to letter case, you can use any combination of upper- and lowercase letters. After typing the last letter of your login name, press the RETURN key. In response, the system displays a password prompt, as shown here:

LOGIN: **jim**
Password:

To protect your password, you will not see what you type when you answer this prompt. (Nor will the cursor move.) After typing the last letter of your password, press RETURN.

Note: If you think you mistyped your password, just press RETURN. The system will give you a new login prompt, so you can start over.

If you correctly typed both your login name and password, your terminal will look similar to this:

LOGIN: **jim**
Password:

Logged in jim Nov-14-1984 09:53:47 on qtty3
Message of the day: Welcome to the Cromix-Plus Operating System
jim[1]

Each time you log in, the Cromix-Plus system displays the message of the day. It announces that you are logged in to the system. Of greater importance, however, is the Shell prompt to the left of the cursor.

The Shell prompt is the login name followed by the command number in brackets. With each command the command number increases by one. The command numbers are used by the Cromix-Plus Command Line Editor (see appendix A).

Because the Shell is your means of communicating with the operating system, you will see the prompt again and again. With the prompt displayed, you can give a command to the Shell.

A command is a typed instruction, which you send to the Shell by pressing RETURN. Until you press RETURN, you can correct or change any command. Use the DELETE key to erase characters; then retype the command. If you prefer, erase the entire command line by pressing CONTROL-U (hold down the CONTROL key and type "U"). Under the Cromix-Plus system, CONTROL-U is a "line-kill" key.

Appendix A, "The Cromix-Plus Command-Line Editor," explains other ways to correct or change a command line.

2.4 Some Conventions Used in this Manual

In text, command names are capitalized, as are the names of corresponding utility programs.

Example:

Time command
Time utility

For each command, there is a sample command line.

jim[1] **time**

To give the command, type what's printed in **boldfaced** type (using any combination of upper- and lowercase letters), and press the RETURN key. *Pressing RETURN sends the command to the system.* Under the Cromix-Plus system, RETURN is an "enter" key.

Almost all the commands in this manual can be given by any user--privileged or nonprivileged. Privileged access is discussed at the end of this chapter.

2.5 Giving Some Typical Commands

Because the Cromix-Plus System is a multi-user system, it is often helpful to know who is currently logged in. You can find out by giving the Who command:

jim[1] **who**

When you press RETURN to enter the command, the Shell calls the Who utility program. ("Calling" a program starts it running.) In response, Who executes, displaying a list of users on the terminal screen. A typical display is shown here:

```
jim[1] who
betty      qtty1      Nov-14-1984 07:16:22      3      7
jim        qtty2      Nov-14-1984 09:42:29      4      1
fred       qtty3      Nov-14-1984 09:53:47      5      1
jim[2]
```

This list of login names and terminal ("qtty") numbers tells you who is currently using the system, from what terminal. Also shown is the time each user logged in to the system.

Note: You can log in from any unused terminal--even if you are already logged in somewhere else. Although the Cromix-Plus System keeps track of the terminal (or terminals) you are using, it does not associate you with a specific terminal.

When a program such as Who is through executing, it returns control to the Shell. The Shell then displays a new prompt so you can give another command. To demonstrate, give the Time command.

```
jim[1] time
Wednesday, November 14, 1984          9:58:12
jim[2]
```

In response, Time displays today's date and the current time on your terminal screen.

2.6 Command Arguments

By typing additional items on a command line, you can modify what a command does. These additional items are called command arguments.

To demonstrate, give this variation of the Who command.

jim[1] **who am i**

This time, Who displays information about *you* only (your login name, your terminal, and the time you logged in). A typical display is shown here:

```
jim[1] who am i
jim          qtty3        Nov-14-1984 09:53:47        4        1
jim[2]
```

Many commands can use arguments. For some commands, arguments are required. For others (such as Who), arguments are optional.

2.7 Command Syntax

The required form of a command--how the command and its arguments are grouped on the command line--is called command syntax. When representing command syntax, the Cromix-Plus documentation shows optional arguments in square brackets.

For example, the syntax of the Who command can be represented as follows:

who [am i]

The command arguments (in brackets) are optional.

A basic understanding of command syntax will help you use the system's on-line manual. There are additional examples of command syntax throughout this manual.

2.8 Displaying the On-line Manual

The Help utility displays "pages" from the system's on-line manual. To display information about a particular command, give the Help command with an argument.

The syntax of the Help command is:

help [command-name]

When you give the command, the argument is the name of any command (such as Who, Time, or

Help).

Figure 2-2 shows a typical display.

```
TIME                      CROMIX Instruction Manual                    TIME

         utility:     TIME
         purpose:     This program displays or alters the time and date.

     user access:     all users for display
                      privileged user for changes

         summary:     time [-se2]

       arguments:     none

         options:     -s   set system values
                      -2   set 3102 clock
                      -e   European style display (dd/mm/yy)

Description

The Time program displays or changes the time and date.  If the -s option
                                                                 75% MORE
```

Figure 2-2: TYPICAL "HELP" DISPLAY

"75% MORE" in the middle of the bottom line of the screen indicates there is more information about this particular command.

Pressing RETURN or the SPACE bar brings additional information onto the screen. Pressing the RETURN key brings a new line into view, while pressing the SPACE bar displays an entire screen of new text.

When you're through using Help, type **q** for *quit*. In a few seconds, the Shell will display a new prompt so you can give another command. While a program like Help is running, you can also get a new prompt by pressing CONTROL-C, as explained in the next section.

Note: Other than CONTROL-C, the remaining keys and/or commands that work with Help (such as pressing RETURN or typing **q**) are specific to Help. Help, like a few other programs, has its own set of commands.

2.9 Stopping a Program While it is Executing

Almost all the commands in this manual execute utility programs like Who, Time, or Help. You can stop most programs while they are executing by pressing CONTROL-C (hold down the CONTROL key and type "C").

Pressing CONTROL-C sends an "abort" signal (via the Shell) to the program. In response, the program prematurely returns control to the Shell. In other words, the program stops running, and the Shell displays a new prompt.

2.10 Displaying Additional Screens of Information

Many programs display output on the terminal screen. Sometimes that output exceeds the capacity of the terminal screen (24 lines). For example, the Query utility displays a summary of Cromix-Plus commands that is longer than 24 lines.

So you can view a program's output at your leisure, the display stops after each screenful. To demonstrate, give the Query command:

jim[1] **query**

After displaying one screenful of text, your terminal beeps. The beep tells you there is more output. To view the next 24 lines of output, press CONTROL-Q (hold down the CONTROL key and type "Q"). Each time you press CONTROL-Q, you'll see a new screen of text. After displaying the final screen of output, Query returns control to the Shell.

This is how your terminal normally displays output. Occasionally, your terminal will display a program's output without stopping the display after one screen.

If this happens, you can restore the normal mode of operation by giving the following command:

jim[1] **mode pa**

This command resets one of your terminal's operating characteristics. The command argument, **pa** (for pause), causes your terminal to, once again, pause after each screen of output.

There is additional information about the Mode utility in the Cromix-Plus User's Reference Manual.

2.11 The Terminal as a Carriage-Return Device

Another of your terminal's operating characteristics is to move the cursor down a line each time you press the RETURN key. This enables you to give commands to the Shell.

Occasionally, without any action on your part, the terminal may not work in this manner. You will press RETURN, and the cursor will not move down a line. To restore the terminal to its normal mode of operation, give the following command:

jim[1] **mode crdev**

To enter the command, you will have to press the DOWN ARROW key. (Pressing RETURN does nothing.) The command argument, **crdev**, re-establishes your terminal as a carriage-return device. After you give the command, pressing RETURN will again move the cursor down a line.

There is additional information about the Mode utility in the Cromix-Plus User's Reference Manual.

2.12 Error Messages

From time to time, all users receive "error messages" from the system. The Cromix-Plus system displays error messages on the terminal screen. Some of these messages come from the Shell; others come from the individual programs.

A common error message is "Command not found"--the message the Shell displays when you give a non-existent command. For example, under the Cromix-Plus system, there is no Chmod utility. Thus, the following command produces an error message:

> jim[1] **chmod**
> Command not found: "chmod"

If you ever mistype a command name, you'll see a similar message when you press RETURN. It means the Shell could not carry out your command. (The Shell looked for, but could not locate, the program.)

Some programs generate error messages of their own. Mode is an example. Although many programs ignore bad arguments, Mode displays an error message:

> jim[1] **mode who**
> Illegal argument: "who"

Some common error messages associated with the Cromix-Plus utility programs are given throughout this manual.

2.13 Changing Your Password

For security, you should occasionally change your password. Whenever you wish to change your password, give the Passwd command:

> jim[1] **passwd**

In response, the Passwd utility prompts for your login name:

> jim[1] **passwd**
>
> Name:

When you type your login name and press RETURN, Passwd displays another prompt. In the example, a user named Jim is changing his password:

jim[1] **passwd**
Name: **jim**
Password:

To answer this prompt, type your *new* password and press RETURN. When you press RETURN, Passwd displays an "encrypted" version of your new password (not what you actually typed).

The system will next ask what your prompt should be:

jim[1] passwd
Name: **jim**
Passwd: xhbydksy
Prompt string:

If the user just types RETURN, the prompt is going to be the user name followed by the command number in brackets. If the user enters some other string that other string will be used as the prompt. The string entered may contain the characters "%d". If so, these two characters will be replaced by the actual command number whenever the prompt is written out. The new prompt will take effect when the user logs in next time.

For the purpose of this example the user just enters RETURN to say he wants to default prompt.

To respond to the next prompt ("Name:"), simply press RETURN:

jim[1] passwd
Name: **jim**
Password: xhbydksy
Prompt string:
Name: (RETURN)
jim[2]

After changing your password in this manner, you must use your new password the next time you log in. (The system no longer recognizes your old password.) If, after several tries, you cannot log in using your new password, see the System Administrator.

2.14 Logging Out

When you are through using the computer, log out by giving the Exit command. Exit, like many commands, can be abbreviated on a command line. To give the Exit command, you can either type the command name in full or a just a portion of it:

jim[1] **ex**

Although the sample command lines in this manual show the abbreviated form for all commands, you can often substitute the full command name (**exit** instead of **ex**).

When you log out, the system redisplays its login prompt. To use the system again from this terminal, you must repeat the login procedure.

2.15 Intrinsic Commands

Exit is an example of an "intrinsic" command. In other words, it is "intrinsic" to the Cromix-Plus System--it is not a utility program like Who or Passwd, which the Shell calls from the computer's hard disk each time you use the program.

Whether the first item on a command line is the name of an intrinsic command (like Exit) or a program (like Who) matters only to the system. The distinction is mentioned to help you understand the Cromix-Plus documentation. For example, the User's Reference Manual summarizes the Shell commands and Cromix-Plus utility programs. "Shell command" means an intrinsic command such as Exit.

2.16 Privileged Access

The system administrator will most likely create your login name to be a nonprivileged user. As such, there are certain commands you cannot give. For example, you cannot use the Passwd utility to change someone else's password--that requires privileged access.

jim[1] **passwd**

Name: **cindy**
Can only change your own password.
jim[2]

You can, of course, change your own password.

As a nonprivileged user, you cannot give a command that might harm the system or inconvenience other users. So, as you use the Cromix-Plus Operating System, feel free to experiment with new commands.

If you accidentally give a command requiring privileged access, the Shell will display an error message, as in the following example:

jim[1] **passwd -n**
Must be privileged user.
jim[2]

Most commands can be given by all users--privileged and nonprivileged.

Chapter 3 - Working With Text Files

In this chapter you'll learn the basic utilities and Shell commands that manipulate files. First, you'll create a sample file using the Cromemco CE program-entry editor. Using this sample file, you can experiment with some of the Cromix-Plus system's most used commands--such as Rename (the command that gives a file a new filename) and Delete (the command that deletes a file from the hard disk).

Also covered is the Spool utility, which prints one or more files. Instructions for stopping printing are included.

3.1 Naming Files

In the next section, you'll create your first file. You can name this file **sample**, as suggested in the text, or select another filename. (In the Cromix-Plus documentation, filenames are always printed in **boldfaced**, lowercase type.) Filenames consist of one to twenty-four *consecutive* characters (in other words, no SPACES) from the following set:

a-z, A-Z, 0-9, _, $, .

Because the Cromix-Plus system is not sensitive to letter case, the following filenames are equivalent:

letters
Letters
LETTERS

You can use the underscore character (_) to make filenames easier to read, as in the following examples:

form_letter
chapter_1

A period in a filename is the first character of a *filename* extension, as in the following examples:

file.text
sort.out

In these filenames, the extensions are **.text** and **.out**. Filename extensions are useful for identifying similar kinds of files:

report.save
chapter2.save
memo.save

To illustrate, you would think twice before deleting a file with a **.save** extension. You might reserve this filename extension for all your important files.

If a filename contains several periods, only the final period (and the characters that follow) is a filename extension.

For example:

file.text.out
sort.out.save

In these filenames, the extensions are **.out** and **.save**. As explained at the end of this chapter, the files you create may not have a **.bin** or **.com** filename extension.

Although you can use a **.bak** (for backup) extension when naming files, its use may prove confusing. The CE program, introduced in this chapter, automatically creates a backup file with a **.bak** filename extension whenever you update an existing file.

3.2 An Additional Consideration When Naming Files

Any Cromix-Plus utility that uses filenames will accept a 24-character filename. Remember, however, that not all the programs you may use are Cromix-Plus utilities.

There are a number of programs that can be installed on the Cromix-Plus system that may have their own file-naming conventions. An example is the Cromemco Formatter-II text-formatting program. Formatter-II is not a Cromix-Plus utility program and will not accept a 24-character filename. (Its limit is 12 characters, including a 3-character filename extension.)

When in doubt, consult the documentation for the program you're using.

3.3 Special Characters

Each of the following characters means something special to the Shell, and should not be used when creating files:

* ? [ ] > < | " ' { } - & ^ \#

In choosing a filename, restrict yourself to those characters shown earlier in the chapter.

3.4 Creating a Sample File

In this section, you'll create a sample file. This manual assumes you'll be using the CE program-entry editor to create the file. CE is another of the Cromix-Plus utility programs, available to all Cromix-Plus users.

In preparation, read the discussion of CE in the *Cromix-Plus User's Reference Manual* where the CE editor is discussed in detail.

Using the CE editor, create a file named **sample**, and type the following text in the file:

```
Help
Passwd
Query
Who
Exit
```

Then save the file.

Because the CE editor has its own set of commands, pressing CONTROL-C does not exit you from the CE program. If you press CONTROL-C with the editor in the Command mode, nothing will happen.

3.5 Displaying a List of Files

The Ls (for list) utility displays useful information about files. Without an argument, Ls displays a simple list of files. To better illustrate the display, the following example shows some typical filenames; obviously, you'll see a different display when you give the command:

```
jim[1] ls
letter      memo      output      plan6_10
```

3.6 Command Options

Command options modify what commands do. On a command line, options follow the command name. They consist of a single alphabetic character, preceded by a hyphen. In the following example, **-m** (a hyphen, followed by the letter "em") is a command option:

```
jim[1] ls -m
239     1 letter
241     1 memo
 50     1 output
240     1 plan6_10
```

The **-m** (for medium) option instructs Ls to display more detailed information about files. From left to right, the display shows you the size of the file (in characters), the number of links to the file, and the filename. (Links are discussed later in this chapter.)

Another option, **-l** (for long), displays even more information:

```
jim[1] ls -l
        239  1 rewa re-- re-- betty          Nov-14 10:18  letter
        241  1 rewa re-- re-- betty          Nov-14 10:18  memo
         50  1 rewa re-- re-- betty          Nov-14 10:22  output
        240  1 rewa re-- re-- betty          Nov-14 10:21  plan6_10
```

Now you can check the time you last worked on a file. You can also see the access privileges associated with each file. (Access privileges and file ownership are discussed in the next section.)

To display still more detailed information about files, give the Ls command with the **-e** (for everything) option. A typical display is shown here:

```
jim[1] ls -e
letter                                239
        created:    Nov-14-1984 10:17:16    rewa re-- re--
        modified:   Nov-14-1984 10:18:08   betty            group: 1
        accessed:   Nov-14-1984 10:17:23   links: 1
        dumped:     000-00-1900 00:00:00   inode: 423

memo                                  241
        created:    Nov-14-1984 10:18:18    rewa re-- re--
        modified:   Nov-14-1984 10:18:55   betty            group: 1
        accessed:   Nov-14-1984 10:18:18   links: 1
        dumped:     000-00-1900 00:00:00   inode: 421

output                                 50
        created:    Nov-14-1984 10:21:55    rewa re-- re--
        modified:   Nov-14-1984 10:22:14   betty            group: 1
        accessed:   Nov-14-1984 10:22:16   links: 1
        dumped:     000-00-1900 00:00:00   inode: 402

plan6_10                              240
        created:    Nov-14-1984 10:19:58    rewa re-- re--
        modified:   Nov-14-1984 10:21:04   betty            group: 1
        accessed:   Nov-14-1984 10:20:30   links: 1
        dumped:     000-00-1900 00:00:00   inode: 422
```

Included in the display is the inode number assigned to each file. (The system uses inode numbers to locate files on the computer's hard disk.) Ls with the **-i** (for inode) option displays only filenames and inode numbers. A System Administrator might give this command to determine the inode number of a particular file:

```
jim[1] ls -i
239  letter
241  memo
```

```
 50  output
240  plan6_10
```

By adding options to a basic command ("Ls"), one utility serves the needs of many--ordinary users, who need to verify filenames, or System Administrators, who need to check inode numbers. Ls is one of many Cromix-Plus utilities that use options.

In command syntax, options are written as in the following example:

```
ls [-eilm]
```

Ls, like a number of Cromix-Plus utilities, will accept multiple options:

```
jim[1] ls -e -m
letter                                239
          created:   Nov-14-1984 10:17:16   rewa re-- re--
          modified:  Nov-14-1984 10:18:08   betty            group: 1
          accessed:  Nov-14-1984 10:17:23   links: 1
          dumped:    000-00-1900 00:00:00   inode: 423

memo                                  241
          created:   Nov-14-1984 10:18:18   rewa re-- re--
          modified:  Nov-14-1984 10:18:55   betty            group: 1
          accessed:  Nov-14-1984 10:18:18   links: 1
          dumped:    000-00-1900 00:00:00   inode: 421

output                                 50
          created:   Nov-14-1984 10:21:55   rewa re-- re--
          modified:  Nov-14-1984 10:22:14   betty            group: 1
          accessed:  Nov-14-1984 10:22:16   links: 1
          dumped:    000-00-1900 00:00:00   inode: 402

plan6_10                              240
          created:   Nov-14-1984 10:19:58   rewa re-- re--
          modified:  Nov-14-1984 10:21:04   betty            group: 1
          accessed:  Nov-14-1984 10:20:30   links: 1
          dumped:    000-00-1900 00:00:00   inode: 422
```

When you give Ls multiple options, the one producing the most extensive list (in this case, **-e**) prevails. If Ls cannot recognize one or more options, a command-syntax summary is displayed so you can see what options are available.

When using a utility that accepts multiple options, you can often group options after a single hyphen, as in this example:

```
jim[1] ls -em
letter                                  239
        created:    Nov-14-1984 10:17:16    rewa re-- re--
        modified:   Nov-14-1984 10:18:08   betty            group: 1
        accessed:   Nov-14-1984 10:17:23   links: 1
        dumped:     000-00-1900 00:00:00   inode: 423

memo                                    241
        created:    Nov-14-1984 10:18:18    rewa re-- re--
        modified:   Nov-14-1984 10:18:55   betty            group: 1
        accessed:   Nov-14-1984 10:18:18   links: 1
        dumped:     000-00-1900 00:00:00   inode: 421

output                                   50
        created:    Nov-14-1984 10:21:55    rewa re-- re--
        modified:   Nov-14-1984 10:22:14   betty            group: 1
        accessed:   Nov-14-1984 10:22:16   links: 1
        dumped:     000-00-1900 00:00:00   inode: 402

plan6_10                                240
        created:    Nov-14-1984 10:19:58    rewa re-- re--
        modified:   Nov-14-1984 10:21:04   betty            group: 1
        accessed:   Nov-14-1984 10:20:30   links: 1
        dumped:     000-00-1900 00:00:00   inode: 422
```

As a general rule, you can group options that do not require arguments. All Ls command options fit this description because none requires additional information on the command line for Ls to act on that option.

General rules regarding command options are summarized at the end of this chapter.

3.7 Access Privileges for the Owner of a File

When you gave the "ls -l" command, you saw a string of lowercase letters and hyphens preceding your login name:

```
jim[1] ls -l
        32    1 rewa re-- re-- you          Nov-14 10:14  sample
```

Your login name means *you* are the owner of the file **sample**. You "own" it in the sense that the Cromix-Plus system created the file for you.

What you can do with the file depends on the access privileges associated with the file. Your privileges (as the owner of a file) are defined by the first four characters in the string of lowercase letters and hyphens. As supplied, the Cromix-Plus system automatically gives you the following access to your files:

rewa

This group of letters:

r for "read"
e for "execute"
w for "write"
a for "append"

means you have all possible access privileges. You can *read* from the file, *write* to the file, *append* (add to the end) of the file, or *execute* the file. Read, write, and append access ensure you can display, change, and add text to files at will. Execute access becomes important when you create command files, as explained in chapter 7.

Having these privileges ensures you can work with your own files. For more information about access privileges, refer to the discussion of the Access utility in the Cromix-Plus User's Reference Manual.

3.8 Displaying the Contents of a File

The Type command displays the contents of text files on the terminal screen. (Any file you can edit using the CE program is a text file.)

To display a file, the command syntax is:

ty filename

For example, you can display the contents of the **sample** file by giving this command:

jim[1] **ty sample**
Help
Passwd
Query
Who
Exit
jim[2]

Type shows you the text in a file. When files are long, pressing CONTROL-Q brings more text onto the terminal screen. Type only displays files--to change the contents of a text file, you must use an editor, such as the CE program.

3.9 Displaying a File with Page Headings and Line Numbers

The Clist utility, like Type, displays the contents of text files. Clist, however, adds some information of its own to the display. In Clist's display, each line of the file is numbered for easy reference. At the top of the display is a heading, showing the name of the file and the last time you worked on it:

```
jim[1] clist sample
File    SAMPLE           Wednesday, November 14, 1984       10:14:29

     1                   Help
     2                   Passwd
     3                   Query
     4                   Who
     5                   Exit
```

If you create text files that contain your own computer programs, you may wish to display those files with Clist. For text files containing letters, memos, and so on, use Type instead.

Neither Type nor Clist change the files they display in any way.

3.10 Making a Copy of a File

When you need another copy of an existing file, use the Copy utility. After a copy operation, you have two files--the original and the copy.

The two files are identical in every respect--except for their filenames. *The Cromix-Plus system will not allow two files to have the same name.*

When you wish to make a copy of a file, the command syntax is:

```
copy source-file destination-file
```

The term "source file" refers to the original file (the source of Copy's input). The term "destination file" refers to the new file. In the Cromix-Plus documentation, and in the on-line help files, you will see these terms often.

Make a copy of the **sample** file now. Call the new file **alpha**.

```
jim[1] copy sample alpha
```

The Ls command verifies the presence of the new file.

```
jim[1] ls
alpha     sample
```

For practice, make a copy of the **alpha** file named **beta**.

```
jim[1] copy alpha beta
```

Using Copy, you created two files that (except for their names) are identical to the file **sample**:

```
jim[1] ty alpha
Help
Passwd
Query
Who
Exit
jim[2] ty beta
Help
Passwd
Query
Who
Exit
jim[3]
```

Giving the "ls -l" command will show that even the creation times associated with the files are the same. If the system created the **sample** file on December 12, at 12:02, the **alpha** and **beta** files will show "Dec-12 12:02" as their creation times, too. Only the inode numbers associated with each file will be different.

3.11 File Links

The Cromix-Plus system identifies files by associating each filename with an inode number. Each inode number corresponds to a physical location on the computer's hard disk. When you create a file, you create both the file itself (under an inode number) and an identifying filename. The filename is also called a link, and every inode must have at least one link. The "ls -m" command

```
jim[1] ls -m
32        1 sample
```

tells you how many links an inode has. In the above example, the "1" indicates that the filename "sample" is the only link to that inode (use the **ls -i** command to get the inode number).

Multiple links are particularly useful for files that have long or awkward pathnames (pathnames are discussed in the next chapter). With the Maklink utility (refer to the Cromix-Plus User's Manual), you can make a link from the original file to a simpler filename in the current directory, and delete it later when you no longer need it. As long as an inode has more than one link, deleting one of the links does not delete the file.

3.12 Renaming a File

The Rename command gives an existing file a new filename.

Command syntax:

ren old-filename new-filename

To demonstrate, give the **sample** file a new filename:

jim[1] **ren sample practice**

The Ls command verifies the file has been renamed:

jim[1] **ls**
alpha beta practice

3.13 Deleting a File

The Delete command deletes files. Once a file is deleted, it no longer appears in the list of files that Ls displays. Nor can you access the information in the file (unless that particular inode had more than one link).

Command syntax:

del file-list

For "file-list," you can substitute a series of filenames. Or, you can delete just one file by giving the command with a single argument (one filename). To demonstrate, delete the file **beta** by giving the following command:

jim[1] **del beta**

You know the file is gone when you see the new Shell prompt.

You can delete several files by giving the command with several arguments (a series of filenames). For example, if you had files named **text**, **demo**, and **chapter**, the following command would delete those files:

jim[1] **del text demo chapter**

Once you delete a file, its contents are irretrievably gone. For this reason, it is a good idea to check

the contents of a file using Type before deleting that file.

3.14 Some Common Error Messages

When using Copy, Delete, or Rename, a new Shell prompt means the system carried out your commands.

jim[1] **copy myfile yourfile**
jim[2] **del myfile**
jim[3]

You know the system *did not* execute a particular command if an error message precedes the new prompt.

jim[1] **ren myfile ourfile**
File not found: "myfile"
jim[2]

This particular message means Rename could not locate the old file (the file to be renamed). Rename returned control to the Shell without renaming a file. To produce a similar error message, you might try renaming the **beta** file--the file you deleted in the previous section.

Some common error messages associated with Rename and Copy are:

jim[1] **ren today**
Wrong number of arguments
jim[2]

This is an example of a syntax error--Rename requires two arguments. Thus, Rename returned control to the Shell without even looking for a file named **today**.

jim[1] **copy letter letter.save**
File already exists: "letter.save"
jim[2]

As a safeguard, Copy and Rename normally require a new filename as their final command argument. Without this protection, you could easily destroy the contents of an existing file.

If you do want Copy to overwrite an existing file, use the **-f** (for force) option:

jim[1] **copy -f letter letter.save**
jim[2]

3.15 Printing a File

When you want to print a copy of a file, use the Spool utility.

To print files, the command syntax is:

```
spool file-list
```

When you give Spool a filename as an argument, Spool sends a copy of that file to the system printer. To demonstrate, print a copy of the **practice** file by giving this command:

jim[1] **spool practice**
jim[2]

The new Shell prompt tells you Spool found the **practice** file and sent a copy of the file to the system printer. If your printer is turned on, and properly loaded with paper, your file will be printed.

Like Delete and Rename, Spool displays an error message when it cannot locate a particular file.

jim[1] **spool pactice**
File not found: "pactice"
jim[2]

Sometimes, when you give the Spool command, another user's job will be printing. In this case, Spool will add your job to a queue--to be printed in its turn.

To print more than one copy of a file, give the Spool command with the **-m** (multiple-copy) option. Using this option, the following command instructs Spool to print three copies of the **practice** file.

jim[1] **spool -m 3 practice**

The **-m** option requires an argument--the number of copies to print. The option and its argument must be separated by spaces, as shown. When you do not use the **-m** option to specify multiple copies, Spool, by default, prints one copy of a file.

When called without a filename as an argument, Spool can be used with still more options. Two of them, **-l** and **-k**, are especially useful. The **-l** (for list) option displays a list of your own print jobs, whether a job is currently printing or in the queue.

betty[1] **spool -l**

	Filename	User	Seq	Dev	Pri	Pages	Lines	Copies	Form
->	memo	betty	3	5:5	5	1	12	1	0

By adding an option, **-a** (for all), to the command, you can display a list of *all* print jobs (other users' jobs, as well as your own):

betty[1] **spool -la**

	Filename	User	Seq	Dev	Pri	Pages	Lines	Copies	Form
->	memo	betty	3	5:5	5	1	12	1	0
	first_draft	jim	4	5:5	5	19	1498	1	0
	budget	jim	5	5:5	5	1	47	2	0

Spool displays an arrow, pointing to the current job (the one that is printing). Your job's Sequence number is a reference if you need to cancel that particular job. *Pressing CONTROL-C after giving the Spool command does not stop printing.*

You can use Spool with the **-k** (for kill) option to cancel any of your own print jobs. To kill a job, while it prints or as it waits in the queue, the Spool command syntax is:

spool -k sequence-#-or-filename

To illustrate, consider the following list of jobs:

jim[1] **spool -la**

	Filename	User	Seq	Dev	Pri	Pages	Lines	Copies	Form
->	memo	betty	3	5:5	5	1	12	1	0
	first_draft	jim	4	5:5	5	19	1498	1	0
	budget	jim	5	5:5	5	1	47	2	0

If Jim wanted to cancel the two jobs he has in the queue, he might give the following commands:

jim[1] **spool -k 4**
jim[2] **spool -k budget**

The first command kills a job by referring to its sequence number, the second, by referring to a filename. Use whichever method you prefer. Be aware, however, that killing a job by sequence number is more precise than killing a job by filename. To illustrate, if Jim had two files named **budget** in the queue, the command "spool -k budget" would kill both of them.

As a nonprivileged user, you can only kill your own print jobs. If Jim tries to kill Betty's job, an error message results:

jim[1] **spool -k 3**
3 not found

The Printer Daemon -- Although it's easiest to call Spool a utility that "prints a file" or "sends a copy of a file to the printer," Spool is a little more complex than this. *Printing is a multi-step process.*

The Spool utility is the process that initiates printing, but it is another process--called the printer daemon--that actually prints your files. Spool activates the daemon before returning control to the Shell.

At your terminal, you see a new Shell prompt, while (behind the scenes) the daemon prints your file.

3.16 General Rules for Command Options

As you use the Cromix-Plus system, you'll encounter numerous programs that can be used with options. The following information will help you use these programs effectively.

1. Option names must be a single character in length.
2. All options must be proceeded by "-".

   jim[1] **ls -m**

3. Options may have arguments.
4. Option arguments cannot be optional.
5. The first optional-argument following an option must be preceded by white space (space or tab).

   jim[1] **readall -c 76 fda**

6. Options with no arguments may be grouped behind one delimiter ("-").

   jim[1] **copy -fv myfile yourfile**

7. Groups of option-arguments following an option must be separated by commas or separated by white space and quoted.

   jim[1] **readall -c 75,76 fda**

   OR

   jim[2] **readall -c "75 76" fda**

8. All options precede command arguments on the command line.
9. "--" may be used to delimit the end of the options. This must be done when the command line arguments could logically be mistaken by the command to be options rather than arguments.

   jim[1] **echo -- -1**

10. The order of options relative to one and other does not matter.

    jim[1] **ftar -cv -b 510 /dev/ftcd .**

is equivalent to:

jim[1] **ftar -b 510 -vc /dev/ftcd .**

11. The order of command arguments may matter, and position-related interpretations should be determined on a command-specific basis.

    jim[1] **copy -v myfile yourfile**

    is NOT equivalent to:

    jim[1] **copy -v yourfile myfile**

12. "-" preceded and followed by white space is used only to mean standard input.

    jim[1] **strcmp - YES OUI SI**

    In this case, **strcmp** will test its command arguments against standard input.

3.17 The .bin, .com, and .cmd Filename Extensions

Three filename extensions have special significance to the Shell -- **.bin**, **.cmd**, and **.com**.

.bin and **.com** mean the file is an executable binary file. For example, the Cromix-Plus utility programs are stored in files with **.bin** extensions. (Files with **.com** extensions were originally designed to work with Cromemco's CDOS Operating System.)

You cannot use the CE program-entry editor (or another text editor) to edit the contents of a binary file.

jim[1] **ce sample.bin**
Cromix-Plus Editor vers. xx.xx
Illegal filename: "sample.bin"

jim[1] **ce sample.com**
Cromix-Plus Editor vers. xx.xx
Illegal filename: "sample.com"

Nor should you display the contents of a binary file using Type or Clist. If you do, your terminal may not work properly until you turn it OFF and back ON.

.cmd means the file is a *command file*. Command files are text files, like the file **practice**. (A text file is any file you can edit with the ce program or display with Type or Clist.) The text in a command file is a series of ordinary commands (like the commands you gave in this chapter).

For example, the following command file contains the "spool -la" command.

```
jim[1] ty sla.cmd
spool -la
```

By giving the Shell the name of a command file--minus the filename extension--you tell the Shell to execute that file.

```
jim[1] sla
```

Executing a command file executes all the commands in the file.

If you're interested in command files, you may wish to duplicate this sample command file. Use the Screen editor to create a file named **sla.cmd**, and proof the file carefully before you save it.

When the Shell executes the file, a list of all print jobs will be displayed--as if you'd given the command "spool -la" in response to the Shell prompt:

```
jim[1] sla
   Filename       User      Seq  Dev    Pri    Pages    Lines  Copies Form
-> chapter1       betty     539  5:5    5      42       6781   1      0
   policy         fred      540  5:5    5      3        132    1      0
```

The system executes the commands in a command file as though you'd given the commands from the terminal keyboard. (To the Shell, you did.) You can write command files to perform many repetitive tasks. Refer to chapter 7 for details.

Before continuing, you may wish to delete some of the sample files you created in this chapter. If so, keep the **practice** file (it is used again in chapter 6).

Chapter 4 - Cromix File Structure

The Cromix-Plus system must keep track of hundreds of files--including executable binary files (such as **who.bin**) and text files (such as **practice**).

In many respects, the Cromix-Plus file system is similar to an ordinary filing system. But, instead of file folders, the Cromix-Plus system organizes information in *directories*. On a computer, information is always stored in files. Thus, the information in a directory is one or more files.

For example, on every Cromix-Plus system, there is a directory called the **bin** directory. In this directory are most files with **.bin** filename extensions (**ce.bin**, **ls.bin**, etc.). The system stores text files (such as **practice**) in other directories. Using the commands in this chapter, you can create some directories and organize your information--the files you create with the Cromix-Plus system.

4.1 The Home Directory

The System Administrator created one directory for you--your user (or home) directory. Normally, the name of your home directory is your login name. Whenever you log in to the system, you are working in your home directory. Unless you change directories (as explained later in the chapter), the system stores your files in your home directory.

In chapter 3, you created a simple file system, or structure. It consists of your home directory and a text file.

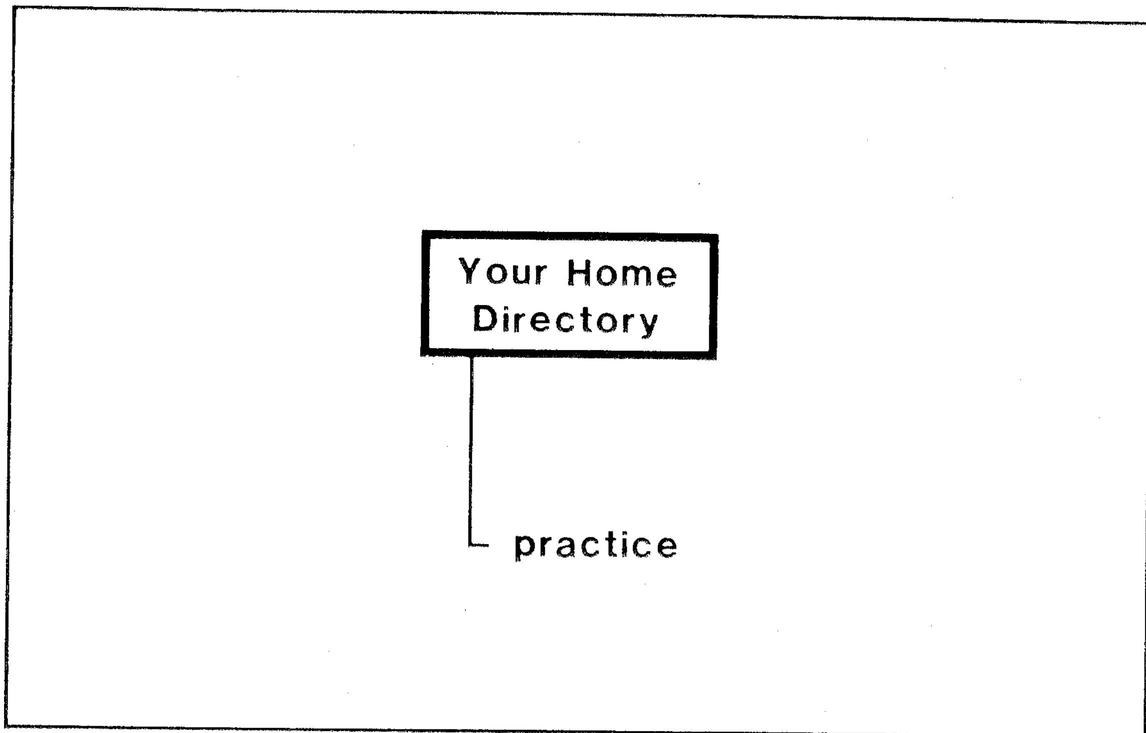

Figure 4-1: FILE STRUCTURE 1

The next section explains how your file structure fits into a much larger structure.

4.2 Visualizing the Cromix-Plus File Structure

The Cromix-Plus file system is hierarchical. In other words, its parts are ranked, so some are on higher levels than others. Figure 4-2 shows a typical system (because there are hundreds of files on any system, only directories are shown):

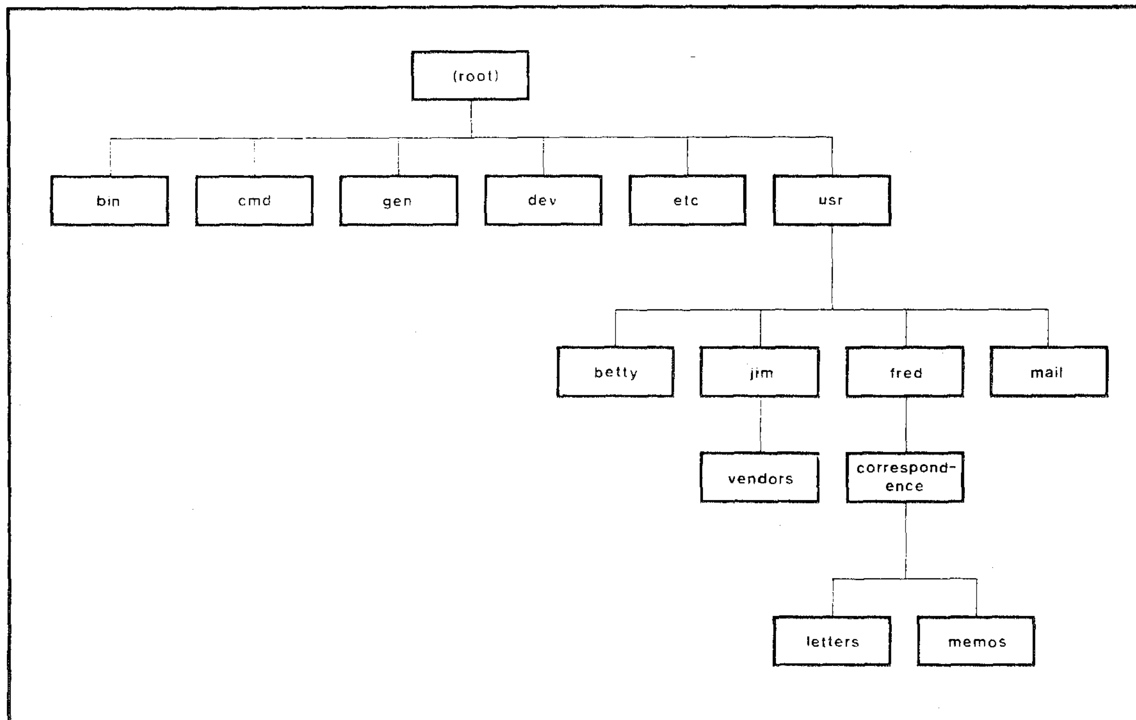

Figure 4-2: FILE STRUCTURE 2

As a family tree traces the ancestry of people, so the Cromix-Plus "tree" traces the ancestry of files. The ancestor of all files is the root directory. Its name is represented by a slash mark (/). Like your home directory, / (root) contains other files. But, most of the files in / are not ordinary files--they are directory files.

To the Cromix-Plus system, each directory (/, your home directory, and so on) is a file. You cannot edit a directory file:

```
jim[1] ce /
Cromix-Plus Editor vers. xx.xx
Not ordinary file: "/"
```

Nor can you execute a directory file:

```
jim[1] /
Command not found: "/"
```

But, directories are files nonetheless. A directory can contain ordinary files, other directory files (called subdirectories), or a combination of ordinary files and subdirectories.

For example, one of the files in / is a subdirectory called **bin**. This directory contains most of the executable files with **.bin** filename extensions. On the same level as **bin** is another subdirectory of / called **usr**. Figure 4-2 shows that **usr** contains subdirectories of its own. One of these subdirectories is your home directory.

Using the analogy of the family tree, / is the parent directory of **bin** and **usr**, while **usr** is the parent directory of every user's home directory.

You can determine the name of your current directory by giving the D (directory) command:

```
jim[1] d
/usr/you
```

Without an argument, the D command prints the absolute pathname of the current directory. Until you change directories, as explained later in the chapter, the command displays the *absolute pathname* of your home directory. It traces a path to the directory file **you** (your login name) from the root directory, / (see figure 4-3).

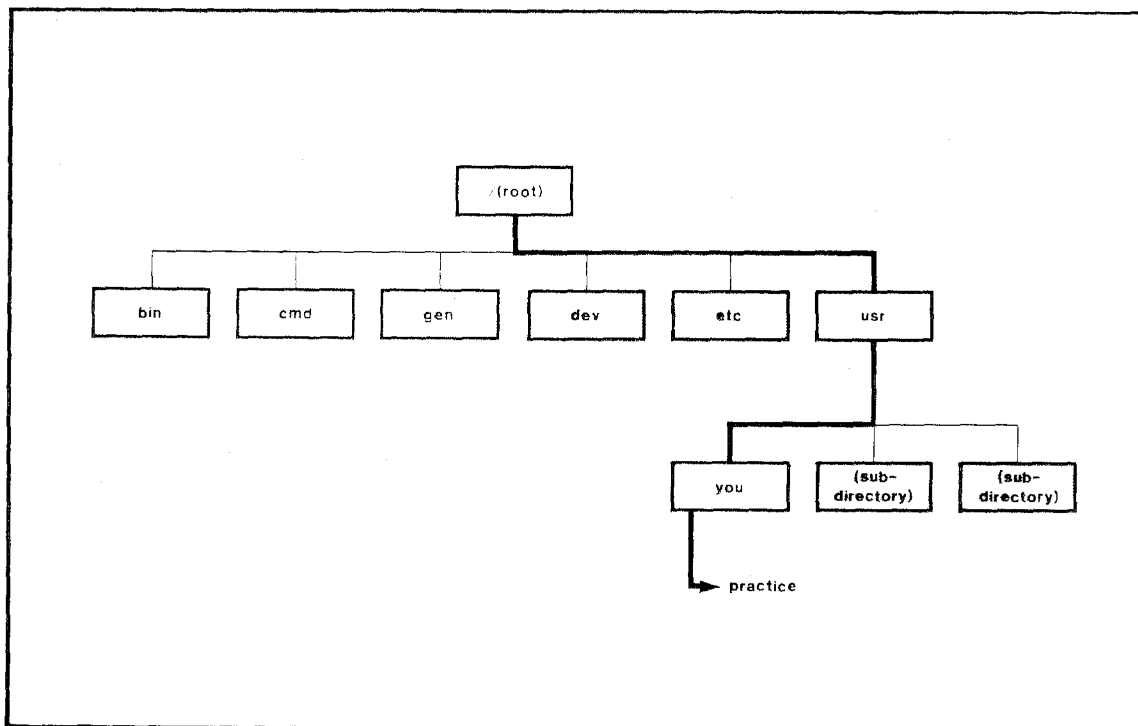

Figure 4-3: FILE STRUCTURE 3

4.3 Absolute Pathnames

The first item in an absolute pathname is the name of the root directory (/). The last item is a filename.

Using absolute pathnames, the system can locate any file in the Cromix file structure. The first "/" tells the system to look for the file from the root directory. Each successive "/" tells the system to look one level deeper in the file structure.

For example, the absolute pathname of the file **who.bin** is **/bin/who.bin**. This pathname gives the system the information it needs to trace a path from the root directory to the file **who.bin** in the **bin** directory. (For clarity, figure 4-4 shows only a few of the files in the **bin** directory.)

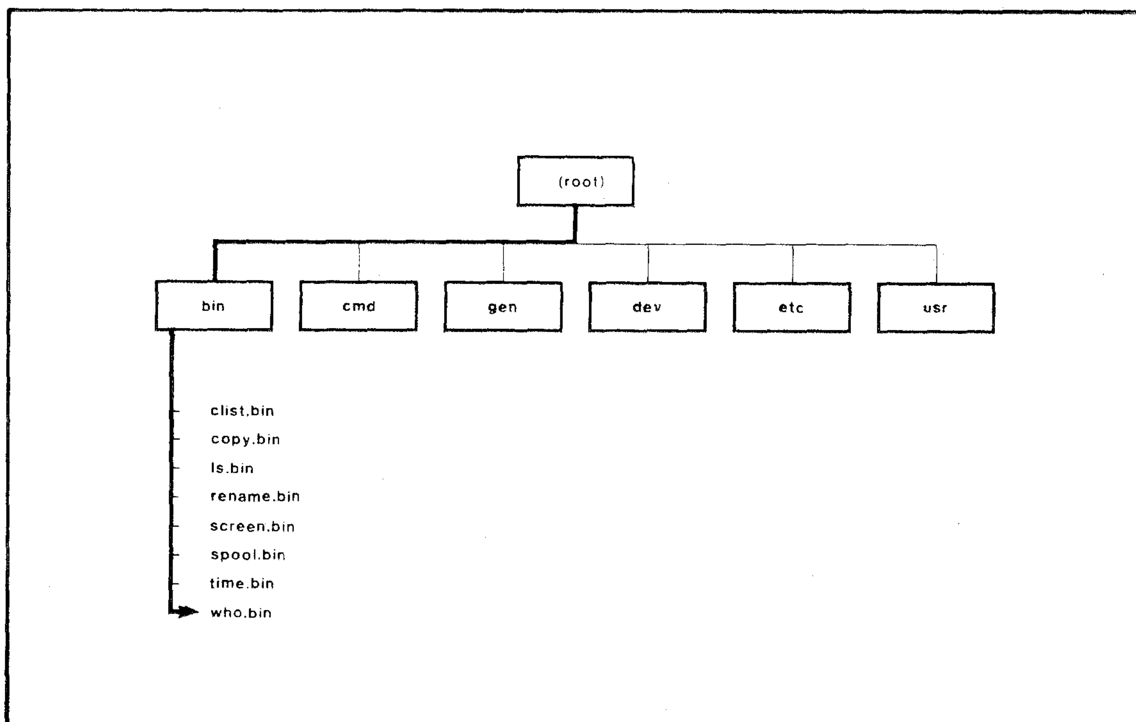

Figure 4-4: FILE STRUCTURE 4

Using the Ls command, you can display information about the files in many directories. Without an argument, Ls displays information for the current directory. With an argument, the pathname of a directory, Ls displays information for that directory.

To illustrate, give the following command:

jim[1] **ls -m /usr**

Without changing the current directory (you're still working in your home directory), Ls displays

information about the files in the **usr** directory. An uppercase "D" means the file is another directory file, a subdirectory of **usr**. Preceding each "D" is the number of files in that directory.

The file corresponding to your login name is your home directory. In the following sample display, **bill** is the name of a user's home directory, as is **alan**, **jimmy**, **mark**, and **sue**.

```
jim[1] ls -m  /usr

Directory: /usr
         6 D     1 alan
         3 D     1 bill
        98 D     1 help
        35 D     1 jimmy
         4 D     1 mail
        27 D     1 mark
         2 D     1 pkg
         5 D     1 query
         4 D     1 spool
        83 D     1 sue
```

Subdirectories such as **help**, **mail**, and **spool** serve other purposes. They are not user directories, as are **mark** or **sue**.

The Ls command with the **-l** or **-e** options displays even more information about the files in **/usr**. (Because the "ls -e" command displays so much information, only a portion of the display is shown.)

```
jim[1] ls -e /usr

Directory: /usr
alan                                      6 directory
        created:   Oct-16-1984 11:13:25      rewa re-- re--
        modified:  Oct-16-1984 11:13:25    alan            group: 1
        accessed:  Nov-14-1984 08:41:14    links: 1
        dumped:    000-00-1900 00:00:00    inode: 393

bill                                      1 directory
        created:   Nov-06-1984 09:59:41      rewa re-- re--
        modified:  Nov-06-1984 09:59:41    bill            group: 1
        accessed:  Nov-06-1984 10:03:49    links: 1
        dumped:    000-00-1900 00:00:00    inode: 157

help                                      98directory
        created:   Oct-12-1984 12:20:36      rewa re-- re--
        modified:  Oct-12-1984 12:20:36    bin             group: 32767
        accessed:  Nov-06-1984 13:27:34    links: 1
        dumped:    000-00-1900 00:00:00    inode: 146
```

.
.

When you give the commands, both displays will show the date and the time the System Administrator created your home directory (**/usr/you**) and the home directories of the other system users.

Ordinary files have absolute pathnames, too. For example, the following command displays the contents of the **practice** file:

jim[1] **ty /usr/you/practice**
Help
Passwd
Query
Who
Exit
jim[2]

The first "/" told the system to look for the file from the root directory. Each successive "/" told the system to look one level deeper in the file structure.

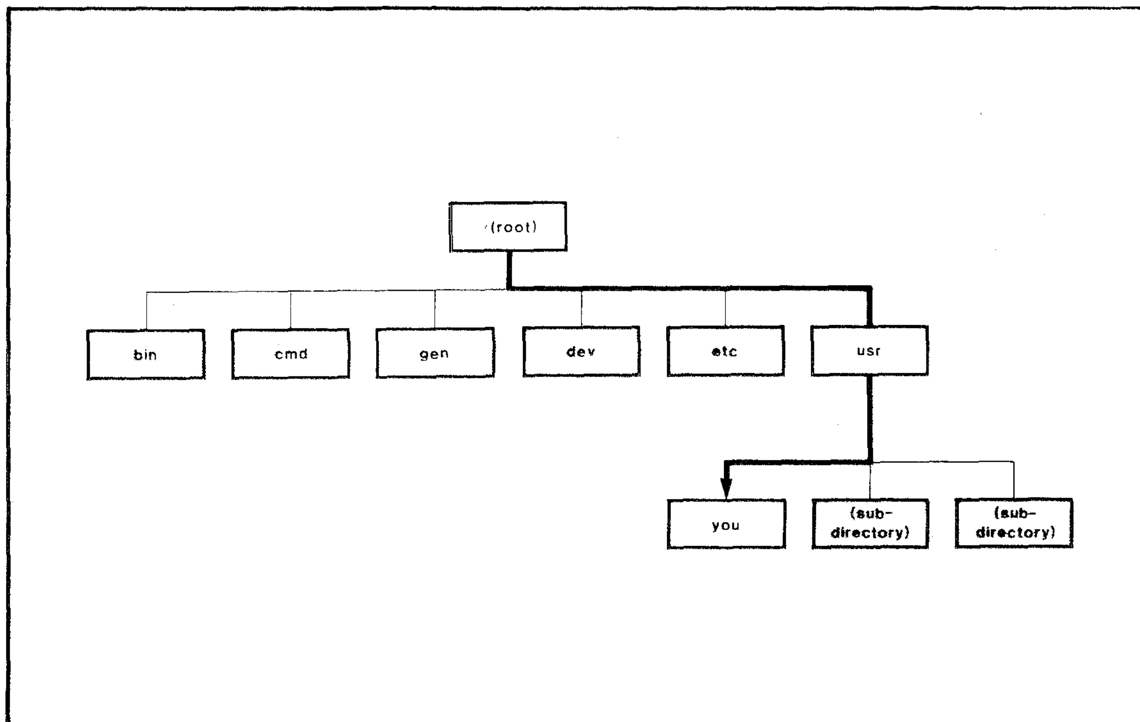

Figure 4-5: FILE STRUCTURE 5

First, the system looked for a subdirectory of / named **usr**. Finding **usr**, the system looked for a subdirectory of **usr** named **you**. Finding **you**, the system looked for (and found) the **practice** file.

Each directory in a pathname must be the parent of the next. For this reason, the following command produces an error message:

jim[1] **ls /etc/usr**
File not found: "/etc/usr"

The system cannot trace a path from the root directory through the **etc** directory to the **usr** directory because **etc** and **usr** are on the same level in the file structure.

As supplied, the Cromix-Plus system makes a few directories inaccessible to nonprivileged users. For example, in the **usr** directory, there is a subdirectory named **mail**. (Chapter 5 explains this directory's purpose.) The access privileges (chapter 3) associated with the file **/usr/mail** do not normally include "execute" access for nonprivileged users. *Without execute access for a directory, you cannot use the name of that directory in a pathname.*

The following command attempts to make **mail** the current directory using an absolute pathname:

jim[1] **d /usr/mail**
Directory not accessible: "/usr/mail"

Because you do have access to the directories you really *need* to work with (such as your home directory), you may never see a similar message. If you do see "Directory not accessible," however, its meaning is always the same. Somewhere in the pathname you gave the Shell is a directory for which you lack execute access.

The System Administrator can explain the situation. If need be, the System Administrator can also change the access privileges for the directory. Refer to the discussion of the Access utility in the Cromix-Plus User's Reference Manual.

A privileged user, such as the System Administrator, automatically has access to all the system's directories.

4.4 How to Make Sure You Have Execute Access for a Directory

You have execute access for many files you do not "own" simply because you can log in to the Cromix-Plus system. When discussing access privileges, anyone who can log in is called a member of the "public".

When you give the "ls -l" command, public access privileges are defined by the last four characters preceding each filename. A lowercase "e" means you, as a member of the public, have execute access for that file.

To illustrate, consider the following display:

```
jim[1] ls -l /usr

Directory: /usr
          6 D 1 rewa re-- re-- alan          Oct-16    11:13    alan
          3 D 1 rewa re-- re-- bill          Nov-06    09:59    bill
         98 D 1 rewa re-- re-- bin           Oct-12    12:20    help
         35 D 1 rewa re-- re-- jimmy         Oct-12    14:40    jimmy
          4 D 1 rewa ---- ---- bin           Oct-12    12:20    mail
         27 D 1 rewa re-- re-- mark          Oct-16    11:20    mark
          2 D 1 rewa re-- re-- bin           Oct-12    12:20    pkg
          5 D 1 rewa re-- re-- bin           Oct-12    12:20    query
          4 D 1 rewa ---- ---- bin           May-04    12:20    spool
         83 D 1 rewa re-- re-- system        Oct-12    15:38    sue
```

Members of the public (Alan, Bill, Jimmy, Mark, and Sue) have execute access for all subdirectories of **/usr** except **mail** and **spool**. The four characters preceding these filenames do not include a lowercase "e" (for execute). Thus, none of this system's nonprivileged users may use the name of either directory in a pathname.

Access privileges affect nonprivileged users only. The System Administrator, and any other privileged user, has access to all files. (For more information about access privileges, consult the discussion of the Access utility in the Cromix-Plus User's Reference Manual.)

4.5 Displaying the Absolute Pathname of an Executable File

Using the Path command you can display the absolute pathname of an executable file.

For example:

```
jim[1] path who
/bin/who.bin
```

OR:

```
jim[1] path ce
/bin/ce.bin
```

If Path's argument is the name of an intrinsic command, Path displays the message "Shell command," as in the following examples:

```
jim[1] path ex
ex:    Shell command
jim[1] path del
del:   Shell command
```

You cannot use Path to display the absolute pathnames of ordinary files, such as **practice**:

> **path practice**
> Command not found: "practice"

There is additional information about the Path command in chapter 7.

4.6 Relative Pathnames

A pathname that does *not* begin with a "/" is a relative pathname. The path traced by a relative pathname starts at (or is relative to) the current directory.

A relative pathname can be as simple as the name of an ordinary file. For example, with your home directory the current directory, the following command displays the **practice** file.

> jim[1] **ty practice**

Using a relative pathname, the system looks for (and finds) the file within the current directory.

Like absolute pathnames, relative pathnames can trace a path through many levels of the file structure. Consider figure 4-6, a typical user's file structure. Like your own file structure, it begins with the user's home directory (**fred**). The home directory is the parent of another directory, **correspondence**, and that directory is the parent of other directories (**letters** and **memos**). The contents of the **letters** and **memos** subdirectories are ordinary files.

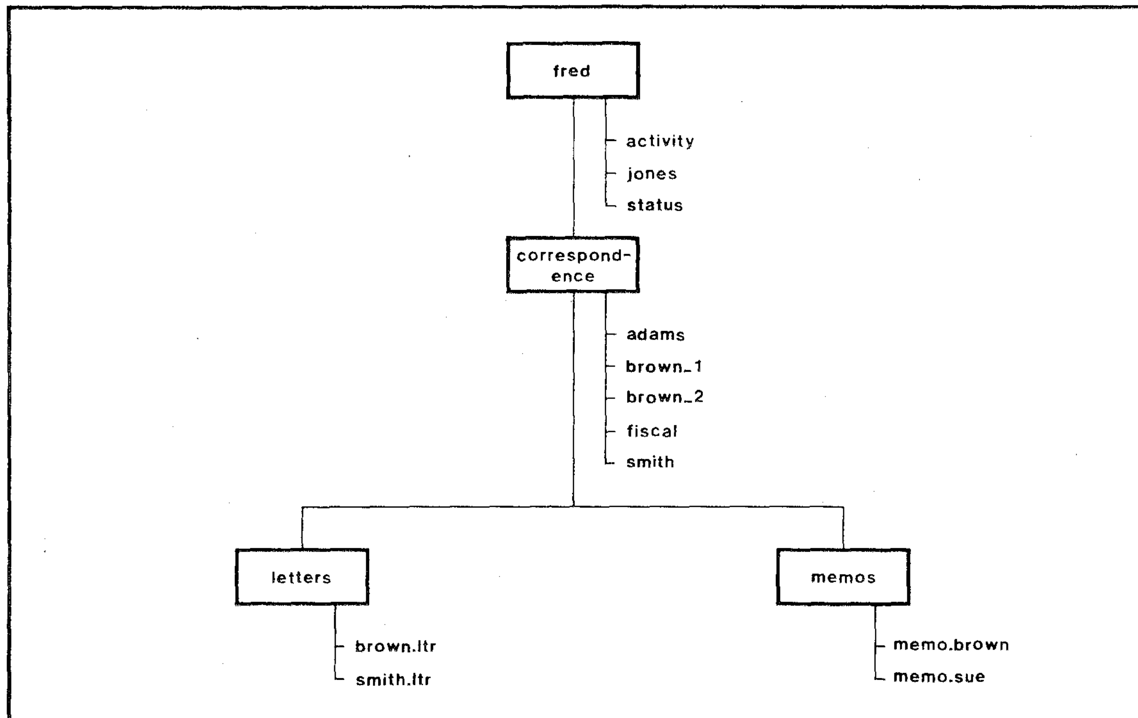

Figure 4-6: FILE STRUCTURE 6

While the home directory is the current directory, the following command displays information about the files in the **memos** directory:

jim[1] **ls -m correspondence/memos**

```
Directory: correspondence/memos
        576     1 memo.brown
        320     1 memo.sue
```

jim[2]

In a similar manner, the next command displays the contents of the file **memo.sue**.

jim[1] **ty correspondence/memos/memo.sue**

Figure 4-7 shows the path the system traced to locate the file.

4.7 Changing Directories

The D command, which you've used to display the absolute pathname of the current directory, can also be used to change directories. To make another directory the current directory, the command syntax is:

d directory-pathname

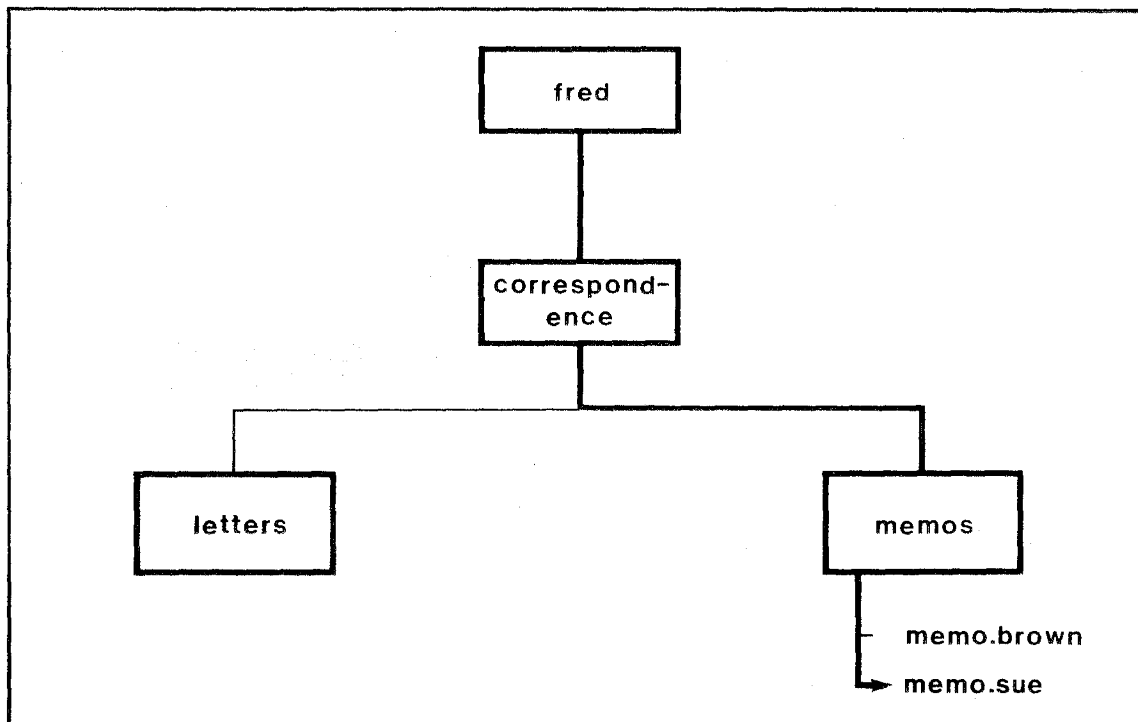

Figure 4-7: FILE STRUCTURE 7

For "directory-pathname," substitute the relative or absolute pathname of the new directory.

To move to any directory that is a *descendant* of the current directory, it's easiest to use a relative pathname. For example, while **fred** (figure 4-6) is the current directory, the following command makes **correspondence** the current directory:

jim[1] **d correspondence**

To move to any directory that is an *ancestor* of the current directory, use an absolute pathname. *When you use absolute pathnames, the current directory is immaterial.* From **correspondence** (or any other directory), the following command makes **fred** the current directory:

jim[1] **d /usr/fred**

In a similar way, while **memos** is the current directory, the following command makes **letters** the current directory:

jim[1] **d /usr/fred/correspondence/letters**

Because the system can locate any file from the root directory, the current directory is immaterial.

Unless you take advantage of the "shortcuts" discussed later in the chapter, you must use absolute pathnames to move *up* in the Cromix-Plus file structure.

4.8 Creating a Directory

The Makd (Make Directory) command creates directory files:

makd directory-pathname

The command argument can be a relative or absolute pathname. Because creating a directory creates a new file, the last item in the pathname must be a *new* filename.

To illustrate, if **fred** (figure 4-6) is the current directory, the following command creates a subdirectory of **letters** named **jan_june**.

jim[1] **makd correspondence/letters/jan_june**

If **letters** (figure 4-8) is the current directory, the pathname is simply a new filename.

jim[1] **makd jan_june**

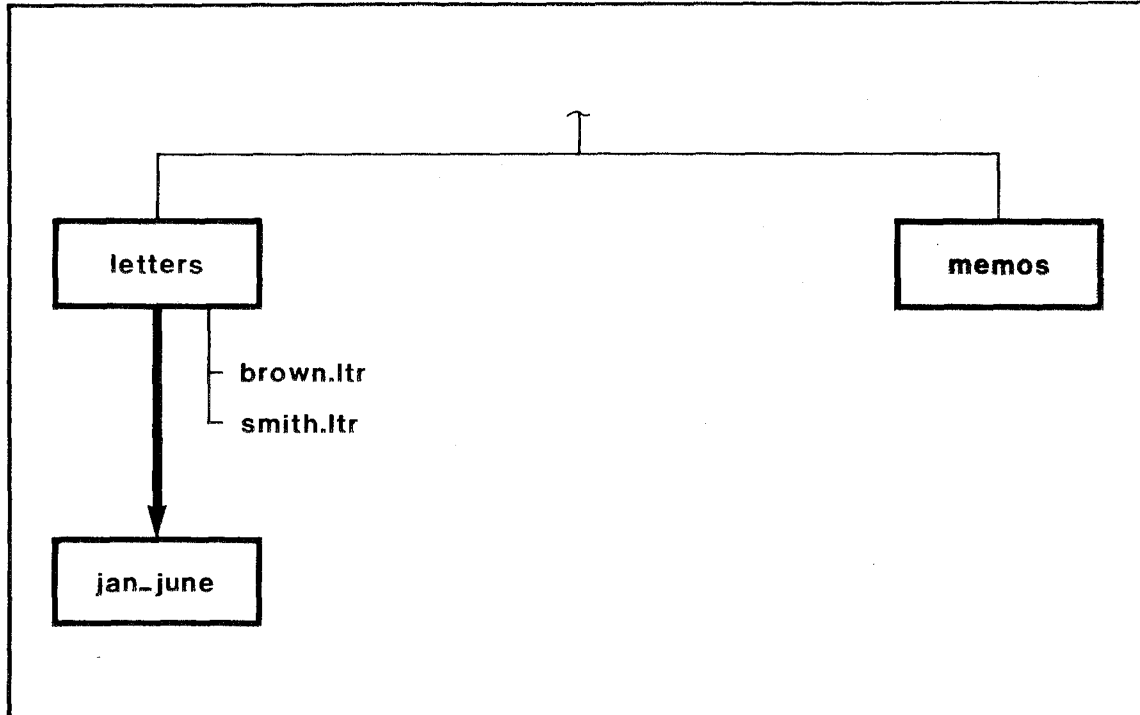

Figure 4-8: FILE STRUCTURE 8

4.9 Moving Files to a Directory

The Move utility moves files from one directory to another:

move file-pathname(s) directory-pathname

When moving a group of files, it is usually easiest to give the command from the directory that contains the files. In this way, you reduce the need for typing long pathnames.

To illustrate, if **correspondence** (figure 4-6) is the current directory, the following command moves two files from **correspondence** to the **jan_june** subdirectory of **letters**:

jim [1] **move brown_1 brown_2 letters/jan_june**

Each filename is the shortest kind of relative pathname.

4.10 How Move Works

In "moving" files within a Cromix file structure, *Move does not physically move files.* A file you "move" is still stored at the same location on the computer's hard disk (the file's inode number is unchanged). The Move command creates a new link to the inode and deletes the old link.

For example, in "moving" **brown_1** to **jan_june**, all Move did was change the absolute pathname of the file from

/usr/fred/correspondence/letters/brown_1

to

/usr/fred/correspondence/letters/jan_june/brown_1

4.11 Copying Files to Another Directory

The Copy command (chapter 3) can also copy one or more files to another directory:

copy file-pathname(s) directory-pathname

For example, while **correspondence** is the current directory, the following command makes a copy of the file **smith** in the **letters** directory:

jim[1] **copy smith letters**

There are now two files named **smith**, one in the **correspondence** directory, another in **letters**:

```
jim[1] ls /usr/fred/correspondence

Directory: /usr/fred/correspondence
adams     fiscal     letters     memos     smith

jim[1] ls /usr/fred/correspondence/letters

Directory: /usr/fred/correspondence/letters
brown.ltr     jan_june     smith     smith.ltr
```

Files in different directories can have the same name. Because their pathnames are different, they are different files to the Cromix-Plus system.

4.12 Renaming Files With Move and Copy

By using another form of these commands, you can rename a file as you move or copy it to another directory:

```
move file-pathname file-pathname
copy file-pathname file-pathname
```

To rename a file, the final command argument is the pathname of an ordinary file (not a directory pathname).

For example, with **correspondence** the current directory, the following command moves a file named **fiscal** to the **memos** directory.

```
jim[1] move fiscal memos/budget
```

Because **budget** is an ordinary filename, the file is renamed **budget** in its new directory, as shown in figure 4-9.

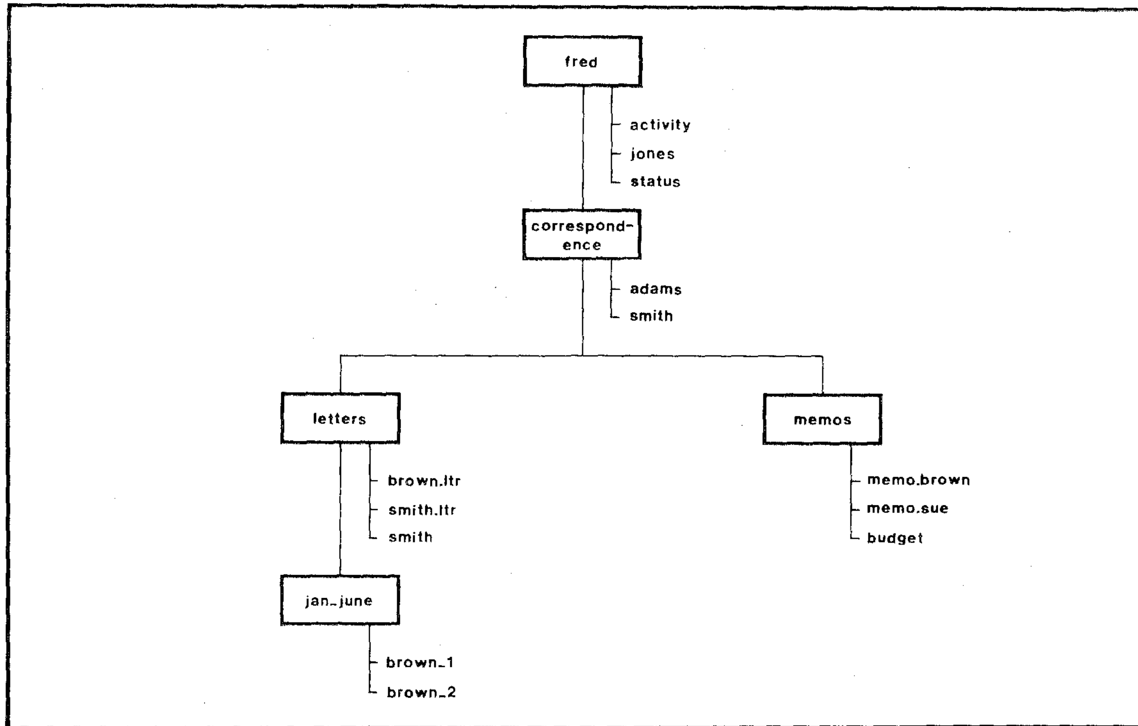

Figure 4-9: FILE STRUCTURE 9

It is possible to accidentally rename files using Move. To demonstrate, suppose that Fred's **correspondence** directory is the current directory, and he gives the following command to move the file **adams** from **correspondence** to **letters**:

jim[1] **move adams lettrs**

Because there is no subdirectory of the current directory named **lettrs**, Move renames **adams** to **lettrs**--within the current directory.

jim[1] **ls /usr/fred/correspondence**

```
Directory: /usr/fred/correspondence
letters    lettrs    memos    smith
jim[2]
```

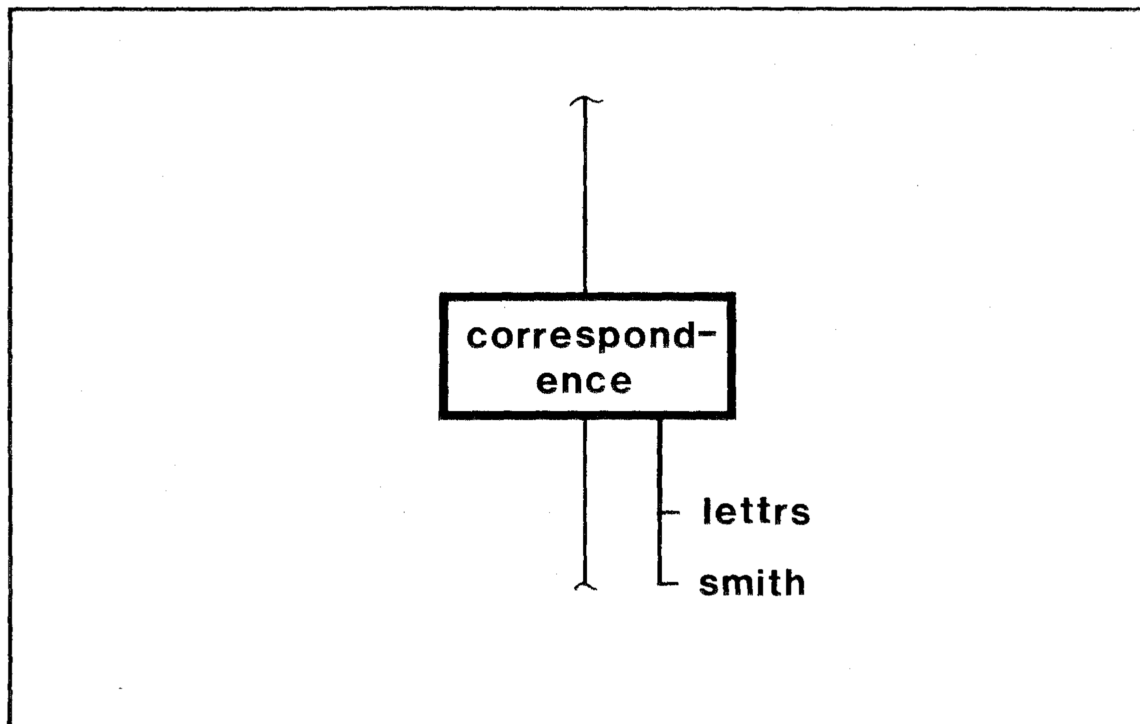

Figure 4-10: FILE STRUCTURE 10

4.13 Shortcuts for Working Within a Directory Structure

To reduce the need for typing long pathnames, the Cromix-Plus system provides some useful notations:

1. A single period (.) represents the current directory.
2. A double period (..) represents the home directory.
3. A caret (^) represents the parent directory.

Using the file structure shown in figure 4-11, here are some things you can do with these notations.

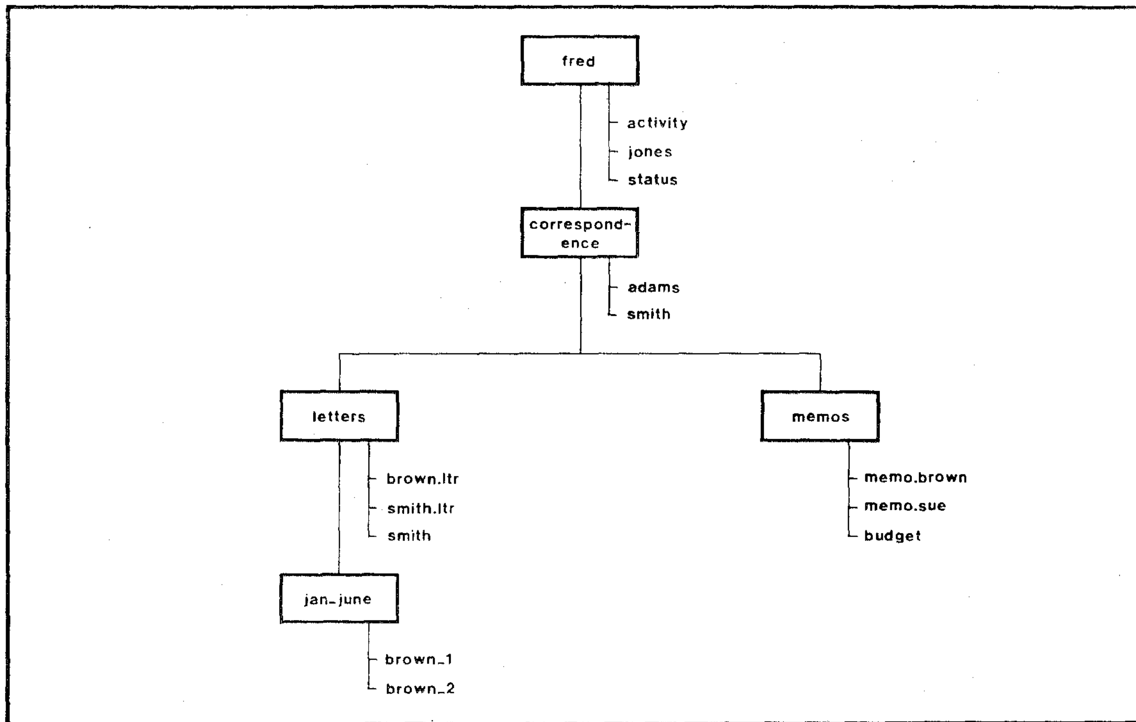

Figure 4-11: FILE STRUCTURE 11

With **jan_june** the current directory, the following command makes **letters** the current directory:

jim[1] **d ^**

With **letters** the current directory, the next command moves a file named **jones** from the home directory (..) to **letters** (.):

jim[1] **move ../jones .**

Without these notations, you would have to use absolute pathnames to move the file:

jim[1] **move /usr/fred/jones /usr/fred/correspondence/letters**

The double period (..) is often used to make the home directory the current directory:

jim[1] **d ..**
jim[2] **d**
/usr/jim

The command "d .." makes the home directory the current directory from *any* directory in the Cromix file structure.

If **jan_june** is the current directory, this command makes its parent directory, **letters**, the current directory:

jim[1] **d ^**

Or, from **jan_june**, this command makes the "grandparent" of **jan_june** the current directory:

jim[1] **d ^^**
jim[2] **d**
/usr/fred/correspondence

Each ^ moves one level higher in the Cromix file structure.

One or more carets (^) can also be used as part of a pathname. To illustrate, if **letters** is the current directory, the following command displays the file **memo.brown** in the **memos** directory:

jim[1] **ty ^memos/memo.brown**

The system started looking for the file from the parent of the current directory.

4.14 Deleting a Directory Structure

You can delete a directory using the Del command (chapter 3), as you would any other file. You must, of course, delete all the directory's descendants (files and subdirectories) before deleting the directory itself. The following series of commands would delete the **memos** directory from its parent directory, **correspondence**.

jim[1] **del memos/budget**
jim[2] **del memos/memo.brown**
jim[3] **del memos/memo.sue**
jim[4] **del memos**
jim[5]

To simplify the process, the Cromix-Plus system provides the Deltree (Delete Tree) utility, which deletes an entire directory structure (or "tree"). Deltree deletes the directory and all its descendants.

The Deltree command syntax is:

deltree directory-pathname

Deltree asks for confirmation before deleting each file. To answer, type **y** or **n** without pressing RETURN.

jim[1] **deltree memos**
Delete memos/budget? **y**
Delete memos/memo.brown? **y**
Delete memos/memo.sue?

Deltree's final prompt asks if the directory itself should be deleted.

jim[1] **deltree memos**
Delete memos/budget? **y**
Delete memos/memo.brown? **y**
Delete memos/memo.sue? **y**
Delete memos? **y**
jim[2]

Because Deltree asks for confirmation before deleting each file, it can be used to "prune" a directory structure. In other words, without intending to delete an entire directory structure, you can use Deltree to selectively delete a series of files.

When you really want to delete all files, you may wish to use the **-a** (for all) option.

jim[1] **deltree -a memos**
Do you really want to delete all of memos? **y**
jim[2]

When you select the **-a** option, Deltree asks if you wish to delete the entire directory structure. If you type **y**, Deltree deletes all files--without requesting further confirmation.

4.15 Copying a Directory Structure

The Cptree (Copy Tree) utility makes a copy of all or parts of a directory structure. Before using Cptree, you must first create a destination directory with Makd. This example creates a destination directory named **oldletters** (the command is given from the **correspondence** directory):

jim[1] **makd oldletters**

To copy an entire "tree" (a directory and all its descendants), the command syntax is:

cptree source destination

"Source" refers to the existing directory and its descendants--the source of the new directory structure.

For example, with **correspondence** the current directory, the following command copies **letters** and all its descendants to the **oldletters** directory.

jim[1] **cptree letters oldletters**

Cptree copies **letters** and all its descendant files, as shown in figure 4-12.

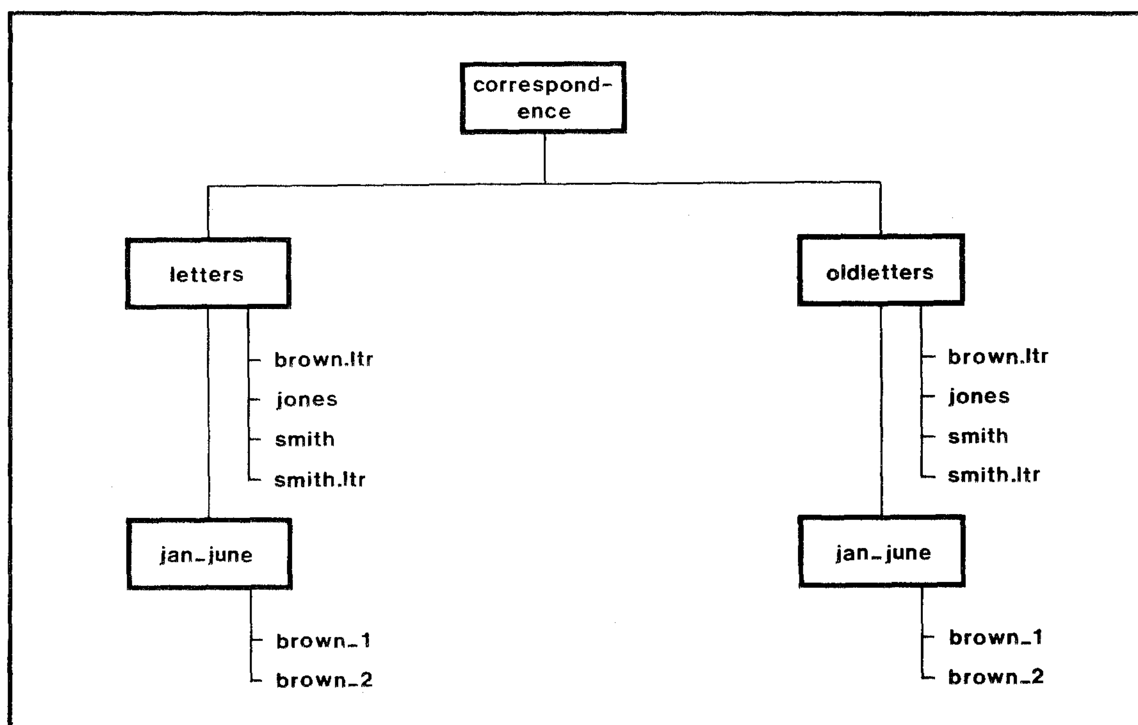

Figure 4-12: FILE STRUCTURE 12

Because it can take a long time to copy an entire "tree," you may wish to use the **-v** (for verbose) option when you give the command. Cptree with the -v option displays each filename as it is copied.

jim[1] **cptree -v letters oldletters**
oldletters/brown.ltr
oldletters/smith.ltr
oldletters/jan_june
oldletters/jan_june/brown_1

oldletters/jan_june/brown_2
oldletters/jones
oldletters/smith

In using the Cromix-Plus system, you'll discover other commands you can give with a **-v** (for verbose) option. Selecting this option displays useful information on the terminal screen while the program executes. The information displayed depends on the program.

4.16 How the Shell Looks for Executable Files

When a command is entered, the Shell interprets the first item on the command line as the name of an executable file. Under Cromix-Plus, all executable files contain either the **.bin**, **.com** or **.cmd** filename extension.

For example:

jim[1] **time**

causes the Shell to look for an executable file named **time**, with a **.bin**, **.com** or **.cmd** filename extension. When the program is found it is loaded into memory and it's execution is begun.

It is not necessary to supply the pathnames of commands. The Shell will automatically search a prescribed list of directories for them. The directories and the order in which they are searched by the Shell is determined by the Shell variable **#path**. It is also possible via the **#ext** variable to instruct the Shell on the order of precedence in which to search for commands with the same name, but with different filename extensions (**.bin**, **.com** or **.cmd**).

Both the **#path** and **#ext** variables are set automatically upon invocation of a new Shell through the use of **sh_env** files. When a new Shell is invoked, usually by logging into the system, the Shell first searches the file **/etc/sh_env** for variable definitions (including **#path** and **#ext**) and then searches the file **../sh_env** if it is present (".." denotes the user's home directory).

When both **/etc/sh_env** and **../sh_env** are present, variable definitions in **../sh_env** will be overlayed upon those defined in **/etc/sh_env**. This means that any variables previously defined in **/etc/sh_env** will be be re-defined by the values specified in **../sh_env**. Any variables not previously defined in **/etc/sh_env** will be created. Any variables defined in **/etc/sh_env** and not re-defined in **../sh_env** will remain as set in **/etc/sh_env**. These files may be edited and changed as required.

As shipped from the factory, the values of **#path** and **#ext** are defined in **/etc/sh_env** as follows:

path: /ram:/bin:/cmd:/usr/bin

ext bin:com:cmd

This means that the directory search path (as defined by **#path**) is as follows:

1. The current directory (nothing to the left of the first colon)
2. The **/ram** directory (if present)
3. The **/bin** directory (standard distribution utilities)
4. The **/cmd** directory (standard distribution command files)
5. The **/usr/bin** directory (user programs)

NOTE: The **/ram** directory should only be present if your system has a RAM disk containing executable files.

The extension search precedence (as defined by **#ext**) is as follows:

1. command**.bin**
2. command**.com**
3. command**.cmd**

Following the search path defined above, the Shell will search for the file in the following way:

1. Within the current directory, the Shell will search for the command name with the **.bin** extension. If **command.bin** is not found, the Shell will search for the same file with a **.com** extension. If **command.com** is not found, the Shell will search for the same file with a **.cmd** extension.
2. If **command.cmd** is not found within the current directory, the search proceeds to the **/ram** directory (if present). Within **/ram**, the Shell will search for **command.bin**, **command.com**, and **command.cmd**.
3. This procedure will be repeated in turn within both the **/cmd** and **/usr/bin** directories. If a file with the command name and any of the three extensions is not found in the prescribed directories, the Shell will display the message:

   Command not found: "command.cmd"

NOTE: Please refer to the "shell" and "set" entries in the *Cromix-Plus User's Reference Manual* for further information regarding Shell variables.

4.17 Special Files in the Home Directory

If you create a file in your home directory named **.reminder**, the Shell will display the contents of that file on the terminal screen whenever you log in to the system.

The contents of a typical **.reminder** file is shown here:

```
jim[1] ty .reminder
Don't forget to fill out your timesheet.
jim[2]
```

Logging in with this **.reminder** file in the home directory might look like this:

```
LOGIN: jane
Password:

Logged in jane     Nov-15-1984    12:18:40 on qtty1
Message of the day: Welcome to the Cromix-Plus Operating System
Don't forget to fill out your timesheet.
jane[1]
```

You may also wish to create a file in your home directory named **.startup.cmd**. The Shell will execute this command file whenever you log in.

A simple **.startup.cmd** file might contain these commands:

```
jim[1] ty .startup.cmd
who
ls
jim[2]
```

With this file in the home directory, logging in might produce the following result:

```
LOGIN: fred
Password:

Logged in fred     Nov-14-1984  09:53:47 on qtty3
Message of the day: Welcome to the Cromix-Plus Operating System
betty      qtty1        Nov-14-1984 07:16:22      0    0
jim        qtty2        Nov-14-1984 09:42:29      0    0
fred       qtty3        Nov-14-1984 09:53:47      0    0
activity        correspondence      jones         status
fred[1]
```

The file **sh_env**, located in your home directory, is a vehicle by which you can have the Shell define Shell variables, including the command search variables **#path** and **#ext**. Please refer to Section 4.16 and the *Cromix-Plus User's Reference Manual* for details regarding Shell variables.

The file **.ce_env**, located in your home directory, will be consulted by the CE screen editor upon invocation. It can be used to customize the editing environment to your tastes. Please refer to the

discussion of CE in the *Cromix-Plus User's Reference Manual* for details.

Because they begin with periods, **.reminder**, **.ce_env** and **.startup.cmd** are "invisible" filenames. The Ls command will not display information about the files unless you use the **-a** (for all) option. In the following example, **-m** and **-a** combine to produce a medium-detailed list of *all* files.

```
fred[1] ls -ma
      9       1 .startup.cmd
    950       1 activity
      5 D     1 correspondence
    505       1 jones
  1,216       1 status
```

4.18 Device Files

The System Administrator often works with files that are neither ordinary files nor directory files. This third kind of file is the device file.

Most device files represent the system's peripheral devices. A peripheral device is any piece of hardware attached to the computer that is not a part of the computer. The system printer is a peripheral device. So is every terminal connected to the computer. Each has a corresponding device file in the **/dev** (device) directory.

You see your terminal's device filename whenever you give the Who command:

```
jim[1] who am i
john       qtty7          Nov-15-1984 11:35:51        0      0
```

In the sample display, it is **qtty7**, or **/dev/qtty7** (the file's absolute pathname).

The System Administrator uses the Makdev utility to add devices to the system. Makdev creates device files. Although you might never work with device files as the System Administrator does, knowing they exist may help you better understand the Cromix-Plus system.

To the Cromix-Plus system, devices and disk files (such as **practice**) appear to be files. This compatibility allows the Shell to redirect input and output as described in chapter 6.

Chapter 5 - The Mail Utility

This chapter discusses the Mail utility--the program that handles the bulk of communication among Cromix-Plus system users. Such communication is from user to user, in the form of typed messages that Mail copies from one terminal for eventual display on another.

A typical mail "delivery" is shown here:

```
----------------------------------------------------------------
From sue, Nov-12-1984 15:57:01

Friday's meeting has been rescheduled. The new
time is Monday at 9:00 A.M.

Let me know if you can attend.
```

The Mail program automatically signs and dates all messages. Thus, whoever receives your message knows who sent it, when, and where to address a reply.

This chapter explains how to send mail and how to read the mail others send you. For more information about communications under the Cromix-Plus system, refer to the discussions of the Mail, Msg, and Ccall utilities in the Cromix-Plus User's Reference Manual.

5.1 Sending Mail

To send mail to another system user, use the following procedure:

1. Give the Mail command with that user's login name as an argument, as in the following example:

   jim[1] **mail john**

2. Type your message, ending each line by pressing RETURN.

    jim[1] **mail john**
    Friday's meeting has been rescheduled. The new
    time is Monday at 9:00 A.M.

3. When you're through, type CONTROL-Z on a line by itself:

    jim[1] **mail john**
    Friday's meeting has been rescheduled. The new
    time is Monday at 9:00 A.M.

    Let me know if you can attend.
    (CONTROL-Z)
    jim[2]

When Mail receives CONTROL-Z (end-of-file) from the terminal, the program knows there is no more input from the terminal. Mail sends the message and returns control to the Shell.

5.2 Correcting Mistakes While Using Mail

As you type your message, you can erase characters on the current line by pressing either the DELETE key or the LEFT ARROW key. After erasing one or more characters, you can then retype that portion of the line. This is the only way to make corrections when giving a program like Mail input from the terminal.

If you want to start over, press CONTROL-C, as you do to stop other programs as they execute. CONTROL-C displays a new Shell prompt--without sending mail.

Note: There is a way you can edit a message you send using Mail as you would an ordinary file. For details, refer to chapter 6, the section "Redirecting Input From a File."

5.3 What Happens to the Mail You Send

When you press CONTROL-Z, your message is not displayed immediately on some other terminal. Instead, it is delivered to a file in the **/usr/mail** directory, where it is stored until it is read.

Each system user has a place for mail within **/usr/mail**. As a nonprivileged user, you probably cannot "list" the files in **/usr/mail** (you lack execute access to the directory). If you're curious about its contents, a typical display is shown here:

system[1] **ls -m /usr/mail**

Directory: /usr/mail
    0   1 alan

```
  0   1 bill
  0   1 jimmy
193   1 mark
  0   1 sue
```

Most of the filenames are user login names. In each of these files, the corresponding user's mail is stored until that user reads the mail from one of the system's terminals.

A "0" preceding the filename means the file is empty (at the moment, that user has no mail). In the sample display, only the user named Mark has mail. There are 193 characters stored in the file **/usr/mail/mark**.

5.4 How Do You Know When You Have Mail

When mail for you is received in **/usr/mail**, the Mail utility notifies you. You will see a message on whatever terminal you're using. Mail even displays the login name of the user who sent you mail, as in the following example:

```
You have mail from sue
```

If you are logged out when you receive mail, you'll see the message "You have mail" as soon as you log in.

Until you read your mail, the system will remind you about it each time you log in. Checking **/usr/mail** is part of the system's login procedure.

5.5 Reading Your Mail

To read mail, give the Mail command without an argument. Here, a user named John logs in to discover he has mail. He then gives the Mail command to read it:

```
LOGIN: john
Password:

Logged in john    Nov-13-1984    08:32:12 on qtty5
You have mail.
Message of the day: Welcome to the Cromix-Plus Operating System
john[1] mail
----------------------------------------------------------------
From sue, Nov-12-1984 15:57:01

Friday's meeting has been rescheduled. The new
time is Monday at 9:00 A.M.

Let me know if you can attend.
```

```
Should mail be saved?
```

To answer Mail's prompt ("Should mail be saved?"), type **y** or **n** and press RETURN. If you type **n** for *no*, Mail discards that item of mail. If you type **y** for *yes*, Mail stores that item of mail in a file named **mbox**.

5.6 The mbox File

The first time you save mail, the Mail program creates a file named **mbox** in the current directory. The file contains the mail you saved.

```
john[1] mail
-------------------------------------------------------------------
From sue, Nov-12-1984 15:57:01

Friday's meeting has been rescheduled. The new
time is Monday at 9:00 A.M.

Let me know if you can attend.

Should mail be saved? y


john[2] ty mbox
-------------------------------------------------------------------
From sue, Nov-12-1984 15:57:01

Friday's meeting has been rescheduled. The new
time is Monday at 9:00 A.M.

Let me know if you can attend.
john[3]
```

The **mbox** file is an ordinary text file. You can edit the file, rename it, print it, and so on. When you have an **mbox** file in the current directory, Mail adds new items of saved mail to the end of the file. If you delete the **mbox** file, Mail will create it again the next time you save mail.

5.7 Sending the Same Mail to Several Users

By giving the Mail command with several arguments (several login names), you can send the same mail to more than one user.

The following example sends the same mail to two users:

jim[1] **mail betty jim**
I need your estimates by Thursday
afternoon. No excuses!
(CONTROL-Z)
jim[2]

When Betty and Jim read their mail, each will receive the same message. By including your own login name on the command line, you can save copies of the mail you send others. The copy is created when you read your own mail and save it in the **mbox** file.

Chapter 6 - The Cromix-Plus Shell

This chapter explains some of the Cromix-Plus Shell's special features. By taking advantage of these features, you can make your work easier. For example, you can let the Shell generate the filenames it passes, as arguments, to a program. When the Shell generates filenames, you don't have to type a long list of filenames on the command line. In a sense, the Shell does it for you.

One of the Shell's most useful features is its ability to redirect a program's output to an ordinary file. When the Shell redirects output to a file, output you would normally see on the terminal is written to a file on the disk. Most of the sample output in this manual, such as the Ls command example shown below, was created by redirecting output to a file:

```
jim[1] ls -m /usr/mail

Directory: /usr/mail
          0        1 alan
          0        1 bill
          0        1 jim
        193        1 mark
          3   D    1 rpkg
          0        1 sue
```

In the example, only the command itself was typed. The rest of the display was produced by redirecting Ls's output to an ordinary file, which was then added to the text of the manual. In this chapter, you will use the Cromix-Plus utilities to create some similar files.

The Shell can also redirect a program's input. Redirected input comes from an ordinary file instead of from the terminal keyboard.

Using another kind of redirection, called a "pipe," the Shell can connect one or more programs specified on the command line. The programs work together to produce a customized result.

To experiment with these, and other, features, you don't need to create a single new file with the CE program. Nor do you need to learn to use additional utility programs. The two new utilities in this chapter (Sort and Create) are introduced only to aid your experiments. You already know enough Cromix-Plus commands to use all of the features discussed in the chapter.

6.1 The Standard Output

The Cromix-Plus system provides a standard file where programs can send their output. It is called STDOUT (for standard output). Although STDOUT is "visible" only to the Shell, you see its effects whenever you use the system.

For example, each time you've given the Who command, you've seen output (a list of system users) on your terminal screen:

```
jim[1] who
betty    qtty1    Nov-14-1984 07:16:22    3    1
jim      qtty2    Nov-14-1984 09:42:29    4    1
fred     qtty3    Nov-14-1984 09:53:47    5    1
```

It happened because the Who utility sends its output to STDOUT, and the Shell connects STDOUT with the terminal screen. Clist, Ls, and Date are some other programs that send output to STDOUT. Their output, too, appears on the terminal screen.

Unless you redirect output as explained in this chapter, STDOUT is always the terminal.

Note: Not all programs use STDOUT. For example, the Spool program sends its output to a system printer--not to the terminal screen.

6.2 The Standard Input

The Cromix-Plus system provides another standard file where programs can get input. It is called STDIN (for standard input). STDIN is connected to the terminal keyboard.

An example of a Cromix-Plus utility that uses STDIN is Mail. When you give Mail an argument (a login name), Mail takes its input from the terminal keyboard.

```
jim[1] mail john
Friday's meeting has been rescheduled. The new
time is Monday at 9:00 A.M.
Let me know if you can attend.
(CONTROL-Z)
jim[2]
```

CONTROL-Z (end-of-file) sends what you typed--via STDIN--to Mail.

Unless you redirect input as explained in this chapter, STDIN is always the terminal keyboard.

Note: Not all programs use STDIN. For example, Who takes its input from a file in the **etc** directory (**/etc/who**)--not from the terminal keyboard.

6.3 The Sort Utility

The Cromix-Plus system provides several programs that can use both STDOUT *and* STDIN. One of them is Sort, which sorts its input (normally, the lines in an ordinary file). In this chapter, you'll use this program to sort the lines in a file.

Your **practice** file (chapter 3), which contains a series of short, unalphabetized lines, is ideal for demonstrating Sort.

```
jim[1] sort practice
Exit
Help
Passwd
Query
Who
```

Sort takes its input--the lines in the **practice** file--and sends sorted output to STDOUT (the terminal screen). That output is sorted alphabetically, as shown above.

Only Sort's output is sorted. The input file (**practice**) is unchanged.

```
jim[1] ty practice
Help
Passwd
Query
Who
Exit
jim[2]
```

Summarized, this is what happened when you gave the Sort command:

1. The Shell located and called the Sort program, passing on the command argument (the filename) to Sort. *The Shell never checks the arguments and command options it passes to a program.*
2. Sort located its input file and sent a sorted version of that file to STDOUT.
3. With STDOUT connected to the terminal, Sort's output appeared on the terminal screen.

A program like Sort, which can use both STDOUT and STDIN, is ideal for experimenting with the redirection techniques discussed in this chapter. The first of these techniques, redirecting output to an ordinary file, is explained in the next section.

Sort is one of the most versatile Cromix-Plus utility programs. For detailed information about this

program, consult the Cromix-Plus User's Reference Manual.

6.4 Redirecting Output to a File

If you add a greater-than symbol (>) and a filename to the command line, the Shell will connect STDOUT to an ordinary file. Then, output that normally appears on the terminal screen is written to a file instead.

```
jim[1] who > who.out
jim[2]
```

The redirect-output symbol (>) instructs the Shell to redirect a program's output from its usual destination--the terminal screen--to a file. In this case, the Shell redirected Who's output to the file **who.out**.

```
jim[1] ty who.out
betty    qtty1    Nov-14-1984 07:16:22    3    1
jim      qtty2    Nov-14-1984 09:42:29    4    1
fred     qtty3    Nov-14-1984 09:53:47    5    1
```

Summarized, this is what happens when you give a command that contains a redirect-output symbol:

1. First, the Shell opens an output file (such as **who.out**) and connects STDOUT with that file. *If the output file already exists, the Shell deletes its contents before it opens the file.*
2. With the output file ready, the Shell calls the program (such as Who).
3. The program executes in response to the Shell. As always, the program sends its output to STDOUT. But, with STDOUT connected to an ordinary file, output that normally goes to the terminal goes to that file instead.

In the following example, Ls's output is redirected to a file named **list.usr**.

```
jim[1] ls -m /usr > list.usr
```

The result is a file containing information about the files in the **usr** directory.

```
jim[1] ty list.usr

Directory: /usr
          6 D 1 alan
          3 D 1 bill
```

```
          98 D 1 help
          35 D 1 jim
           4 D 1 mail
          27 D 1 mark
           2 D 1 pkg
           5 D 1 query
           4 D 1 spool
          83 D 1 sue
jim[2]
```

Note: *Redirecting output to a file works only with programs that send output to STDOUT.*

Sort uses STDOUT. Thus, the following command creates an output file containing a sorted version of **practice** (Sort's input file).

jim[1] **sort practice > practice.out**

The output file **practice.out** is an ordinary file. You can print it using Spool or edit it using CE:

jim[1] **spool practice.out**
jim[2] **ce practice.out**

When you redirect a program's output to a file, the Shell will overwrite any file in the same directory with the same name. To demonstrate, create a file named **test** by giving the following command:

jim[1] **who > test**
jim[2]

The file **test** contains the redirected output from Who.

jim[1] **ty test**

```
betty    qtty1    Nov-14-1984 07:16:22    3    1
jim      qtty2    Nov-14-1984 09:42:29    4    1
fred     qtty3    Nov-14-1984 09:53:47    5    1
jim[2]
```

Now try redirecting the output from Time to the same file. Then display the contents of the file with Type.

```
jim[1] time > test
jim[2] ty test
Wednesday, November 14, 1984                9:58:12
jim[3]
```

The file **test** now contains only the output from Time. In opening a new output file named **test**, the Shell automatically deleted the previous contents of the **test** file.

When you don't want the Shell to overwrite a file with the same name, use the redirect-and-append symbol discussed in the next section.

Important: In opening an output file, the Shell automatically deletes the contents of an existing file with the same name. *This happens BEFORE the Shell calls the program.*

It is thus possible to accidentally delete a file while redirecting output. Consider the following command:

```
jim[1] clist prog.c > prog.c
```

The input file and the output file have the same name (**prog.c**). In executing this command, the Shell deletes the contents of **prog.c** *before* calling Clist. When Clist executes, the file **prog.c** contains nothing.

When redirecting output, make sure the input file (if any) and the output file have different filenames.

6.5 Appending Output to a File

The Shell will not overwrite a file with the same name if you use a redirect-and-append symbol (>>). Instead, the Shell adds (appends) the new output to the end of the existing file.

To demonstrate, give the following commands:

```
jim[1] time >> time.out
jim[2] ls >> time.out
```

The first command created a new file named **time.out**, a file containing Time's output. The second command appended Ls's output to the end of the file.

```
jim[1] ty time.out
        Friday, November 16, 1984           16:28:24
```

```
list.usr    practice    test        time.out    who.out
jim[2]
```

6.6 Redirecting Type's Output to a File

Unless you specify an input file, Type waits for input from the terminal (STDIN). By redirecting Type's output to a file, as you supply input from the terminal keyboard, you can create a text file using Type.

If you'd like to try it, give the following command:

jim[1] **type > type.out**

In response, the Shell calls Type--without passing a filename to the program. (The Shell opens the output file before calling the program.) Because the Shell called Type without an argument, Type expects input from STDIN (the terminal keyboard).

Enter two or three lines of text (as if you were using Mail). When you're through, type CONTROL-Z (for end-of-file).

jim[1] **type > type.out**
This is some sample text
I am typing into a file
named type.out.
(CONTROL-Z)
jim[2]

The new prompt means Type's output has been redirected. You now have a file named **type.out** in your directory.

jim[1] **ls type.out**
type.out
jim[2]

You can create a similar, *sorted* file using Sort:

jim[1] **sort > sort.out**
Music
Message
Case

```
Countess
(CONTROL-Z)
jim[2] ty sort.out
Case
Countess
Message
Music
jim[3]
```

Sort and Type can create files in this way because both programs use STDOUT, and both can take input from STDIN (the terminal keyboard). When called with an argument, a filename, they take their input from a file; when called without an argument, they take their input from STDIN. Some other programs that use STDIN when called without an argument are: Clist, Dump, Match, Scan, and Spool.

6.7 Redirecting Input From a File

If you add a less-than symbol (<) and a filename to the command line, the Shell will connect STDIN to an ordinary file. The redirect-input symbol (<) instructs the Shell to redirect a program's input so it comes from an ordinary file instead of from the terminal keyboard. In this example, the Shell redirects Mail's input from the file **info**.

```
jim[1] mail sue < info
jim[2]
```

Mail sends a copy of the text in the file--as if that text had been typed after giving the Mail command. Summarized, this is what happens when you give a command that contains a redirect-input symbol:

1. First, the Shell locates the input file (such as **info**) and connects STDIN with that file. If the Shell cannot locate the input file, the Shell displays the message "File not found" before calling the program.
2. The Shell calls the program (such as Mail).
3. The program executes in response to the Shell. As always, the program gets its input from STDIN. But, with STDIN connected to an ordinary file, input that normally comes from the terminal comes from that file instead.
4. The program returns control to the Shell when it encounters an end-of-file indicator. (At this point, the Shell displays a new prompt.)

When you redirect a program's input from a file, the Shell supplies the end-of-file indicator for you. You do not need to type CONTROL-Z (for end-of-file), as you do when a program's input comes from the terminal keyboard.

Note: *Redirecting input from a file works only with programs that can take input from STDIN.*

6.8 Running a Job in the Background

So far, your commands have executed in seconds. After each command, the Shell quickly displayed a new prompt so you could give another. Some commands, however, can take much longer to execute.

To illustrate, give the following command:

jim[1] **ls -e /bin > bin.out**

The Shell will not display a new prompt until Ls's output (information about the files in the **bin** directory) has been redirected. Because there are a lot of files in the **bin** directory, and the **-e** option produces so much information, the process will take some time.

When you have a new prompt, try a similar version of the same command. Add an ampersand (&) to the command line:

jim[1] **ls -e /bin > bin.out &**

An ampersand instructs the Shell to start a detached process--a process that is no longer connected to your terminal. In seconds, the Shell displays a PID (process identification) number and a new prompt. You can now give a new command *while* the Ls program executes in the background (as a detached process). You can even log out while the program executes.

Any program that takes a long time to execute can be run in the background. For example, if you are copying a long file, you may wish to run the job as a detached process:

jim[1] **copy longfile longfile.save &**
PID=1578
jim[2]

Using Cptree (chapter 4) to copy an entire directory structure is also time-consuming.

jim[1] **cptree bigdirectory newdirectory &**
PID=5781
jim[2]

While a program runs as a detached process, pressing CONTROL-C will not stop it. To stop a detached process, use the Kill command. Kill's argument is the PID number of the process. This example starts and stops a typical detached process:

```
jim[1] copy bigfile bigfile.copy &
PID=4137
jim[2] kill 4137
jim[3]
```

As a non-privileged user, you can kill any process the Shell starts for *you*. You cannot kill a process the Shell starts for another user.

Important: When the Shell runs Spool as a background process, the PID # you see on the terminal is Spool's process number. The printer daemon (chapter 3) is a separate process, and the Kill utility will not stop it. *To kill background printing, use the "spool -k" command.*

6.9 Giving Sequential Commands

You can give the Shell more than one command on a single command line by separating the commands with a semicolon (;). To illustrate, give the following command sequence:

```
jim[1] time;who;ls
```

In response, Time executes, followed by Who, followed by Ls. The Shell processes each command in the order it appears on the command line. This is called sequential processing.

In the next example, the Shell deletes a file before Ls executes.

```
jim[1] del who.out;ls
list.bin    list.usr    practice    sort.out    test    time.out    type.out
jim[2]
```

When giving the Shell a series of commands, you can also use an ampersand to separate the commands. The Shell executes a command followed by an ampersand as a detached process (in the background).

This example redirects output to a file named **logged_in** in the background and, at the same time, calls the CE editor:

```
jim[1] who > logged_in&ce practice
```

Or, this command starts two detached (or background) processes:

the contents of a file using Type before deleting that file.

3.14 Some Common Error Messages

When using Copy, Delete, or Rename, a new Shell prompt means the system carried out your commands.

jim[1] **copy myfile yourfile**
jim[2] **del myfile**
jim[3]

You know the system *did not* execute a particular command if an error message precedes the new prompt.

jim[1] **ren myfile ourfile**
File not found: "myfile"
jim[2]

This particular message means Rename could not locate the old file (the file to be renamed). Rename returned control to the Shell without renaming a file. To produce a similar error message, you might try renaming the **beta** file--the file you deleted in the previous section.

Some common error messages associated with Rename and Copy are:

jim[1] **ren today**
Wrong number of arguments
jim[2]

This is an example of a syntax error--Rename requires two arguments. Thus, Rename returned control to the Shell without even looking for a file named **today**.

jim[1] **copy letter letter.save**
File already exists: "letter.save"
jim[2]

As a safeguard, Copy and Rename normally require a new filename as their final command argument. Without this protection, you could easily destroy the contents of an existing file.

If you do want Copy to overwrite an existing file, use the **-f** (for force) option:

jim[1] **copy -f letter letter.save**
jim[2]

jim[1] **move memo1 memo2 memo3 memos** >* **error**
jim[2] **move letter1 letter2 correspondence** >>* **error**
jim[3]

Error messages, if any, are contained in the file **error**.

jim[1] **ty error**
File not found: "memo3"
File not found: "letter1"
jim[2]

6.12 Redirection with Pipes

A pipe symbol (|) on the command line instructs the Shell to connect the output from one program to the input of another. For example, the following command uses a pipe to connect Ls with Spool:

jim[1] **ls /usr | spool**

Spool's input is the output from Ls. The result is a printed list of the files in the **/usr** directory, as if you'd given the following commands:

jim[1] **ls /usr** > **list.out**
jim[2] **spool list.out**
jim[3] **del list.out**

A pipe does not create an intermediate file. Instead, Ls's output is channeled directly to Spool.

Try this similar command to connect Who and Sort:

jim[1] **who | sort**

The result is a sorted list of system users.

This command uses *two* pipes to produce a printed, sorted list of system users.

jim[1] **who | sort | spool**

Using pipes, the Shell can connect a variety of programs that were not specifically designed to work together. At the beginning of a pipe is a program that sends output to STDOUT. In the preceding

command, both Who and Sort fill this requirement. At the end of a pipe is a program that can take its input from STDIN. In the sample command, both Sort and Spool fill this requirement.

When you use a pipe (|), the programs run concurrently. The Shell starts three processes (one for each program) when you give this command:

jim[1] **who | sort | spool**

Even though Spool must wait for Who and Sort to execute before it has output to print, it is running nevertheless. If system memory is at a premium, you may wish to use the redirection technique discussed in the next section instead of a pipe.

6.13 Redirecting Output to a Temporary File

Using a redirect-output symbol immediately followed by a redirect-input symbol, you can duplicate the effects of a pipe. For example, the following command prints a list of system users:

jim[1] **who >< spool**

A "><" symbol, called a "sequential pipe," instructs the Shell to create a *temporary* file. In the sample command, the Shell redirects Who's output to a temporary file, and then redirects Spool's input from the same file. The file is "temporary" because the Shell automatically deletes it before displaying a new prompt.

The Shell runs each process (for example, Who and Spool) sequentially. In other words, Who executes before the Shell calls Spool. This form of redirection, although slower than using a pipe, saves system memory.

6.14 The Tee Command

Using the Tee command, you can redirect output to a file and have that output appear on the terminal as well. For example, this command redirects Sort's output to a file (**practice.sort**) and displays that output, too.

jim[1] **sort practice | tee practice.sort**

The Tee command requires an argument--the pathname of an ordinary file. For more information about the Tee command, refer to the Cromix-Plus User's Reference Manual.

6.15 Filename Generation

A command argument that contains an asterisk (*) or question mark (?) is an ambiguous file reference:

1. An asterisk (*) matches a string of zero or more characters. An asterisk *does not* match a leading or embedded period.
2. A double asterisk (**) matches zero or more characters, including an embedded period. A double asterisk *does not* match a leading period.
3. A question mark (?) matches any single character other than a leading period.

In response to an ambiguous file reference, the Shell generates the names of specific files and passes those filenames to a program. For example, suppose you have the following files in your directory:

account
data_sheet
intro
memo
memos
memo.fred
memo1
memo2

The following command displays information about *all* the "memo" files:

jim[1] **l m****
memo memos memo.fred memo1 memo2

Because a double asterisk matches zero or more characters (including an embedded period), the Shell generates the names of all files that begin with "m".

This command displays information about the file **memo.fred** only:

jim[1] **l m*.***
memo.fred

Only this file has a name that consists of "m", followed by zero or more characters, followed by a period, followed by zero or more characters.

This command uses a question mark to display information about the files that begin with "memo", followed by any single character:

jim[1] **l memo?**

memos memo1 memo2

The Shell does not generate the filename **memo** because it does not consist of "memo" followed by another character.

You can usually use an ambiguous file reference wherever you'd use an ordinary filename. In this example, an ambiguous file reference is used to print the file **memo1**:

jim[1] **spool m*1**

Or, in this example, an ambiguous file reference is used to move the files **memos**, **memo1**, and **memo2** to another directory.

jim[1] **move memo? correspondence**

Because ambiguous file references can produce some unforseen matches, be careful when you use them. When in doubt, test them with the Ls command:

jim[1] **ls ?.txt**
1.txt 2.txt 3.txt 4.txt 5.txt

If none of the filenames in a directory begins with a period, a double asterisk--by itself--matches every filename in the directory. Thus, the following command will usually delete all the files in the current directory:

jim[1] **del ****

This command is mentioned because, sooner or later, almost everyone tries it. Resist temptation, and use the Deltree utility instead. Deltree requires confirmation from you before deleting files--Delete does not.

It is safest not to use ambiguous file references with the Delete command. If you feel you must, *always check the current directory* before giving the Delete command.

6.16 Specifying a Range of Characters

Using another kind of ambiguous file reference, you can instruct the Shell to look for a range of characters. The Shell will substitute characters in square brackets for the corresponding character in a filename.

To illustrate, suppose your directory contains the following files:

a_letter
b_letter
c_letter
a_memo
b_memo
c_memo
d_memo

The following command would display information about the files **a_letter** and **b_letter**:

```
jim[1] ls [ab]_letter
a_letter     b_letter
```

Without the brackets, the Shell would pass the argument "ab_letter" to Ls, and Ls would look for an exact match:

```
jim[1] ls ab_letter
file not found: "ab_letter"
```

The letters "a" and "b", in square brackets, define a range of characters the Shell may substitute for the first character in the filename.

In a similar manner, the next command displays information about all files that begin with "a" or d":

```
jim[1] ls [ad]*
a_letter     a_memo     d_memo
```

To specify a wider range of characters, you can use a hyphen within the brackets, as in this example:

```
jim[1] ls [b-d]_memo
b_memo     c_memo     d_memo
```

The hyphen means "through," as it does in normal usage.

6.17 An Important Consideration Regarding Filename Generation

The number of filenames the Shell can generate in response to an ambiguous file reference is limited by the number of characters the Shell can pass, as command arguments, to any program. *The maximum is 512 characters.* When asked to pass more characters than this to any Cromix-Plus program, the Shell displays the message "Arg list too big," as shown below:

jim[1] **ls /bin/****
Arg list too big

If you give this command, you'll see the same message, followed by a partial list of files in the **/bin** directory. This message comes from the Shell. The program (in this case, Ls) has no way of knowing it received a partial list of filenames from the Shell.

If, as you use the Cromix-Plus system, you see an "Arg list too big" message, try breaking the list into two commands, as in the following examples:

jim[1] **move [a-k]** /usr/jerry/sales_leads**
jim[2] **move [l-z]** /usr/jerry/sales_leads**

Better still, avoid the problem by limiting the size of your directories. If the Shell cannot generate the names of all the files in a user's directory, that directory contains too many files. Create some subdirectories, and reduce the clutter.

6.18 Experimenting with Filename Generation

Without a group of files with similar filenames, it's difficult to appreciate the Shell's ability to generate filenames. If you would like to experiment with filename generation, there is an easy, quick way to create a series of files.

The Create command with a filename as an argument creates a file containing zero bytes. The file has a name, but there is no information in the file. By giving the command with a list of filenames, you can create a series of zero-byte files:

jim[1] **create big bigger biggest 2big really_big**

The preceding command will create five new files in your current directory. Although the files contain no characters, they will show up in the list Ls displays. Because their filenames are similar, they are ideal for demonstrating filename generation.

Chapter 7 - Writing Command Files

In addition to general information about command files, this chapter discusses some commands that are often used in command files. A few of these commands (such as Shift and Rewind) are used nowhere else.

Most likely, when you've used the Cromix-Plus system for a while, you'll discover some repetitive task you could do more easily with a command file. By taking advantage of the Shell's ability to interpret command files (the programs you write for the Shell), you can make your work a lot easier.

7.1 Command-File Description

Command files are ordinary files, which contain a series of Cromix-Plus commands. A command file must have the filename extension **.cmd** (for command). The following command file (**list.cmd**) contains the "ls -m" command:

```
jim[1] ty list.cmd
ls -m
```

Giving the command filename (minus the filename extension) from the command line executes the file:

```
jim[1] list
239    1 letter
241    1 memo
 50    1 output
240    1 plan6_10
```

Command files can be this simple. Command files can also perform more complex tasks, such as copying files from the computer's hard disk onto floppy diskettes for long-term storage.

When you give a command that executes a command file, that file should be located in the current directory or the **/cmd** directory. (If **/ram** is present, command files may also reside in that directory.) To execute a command file in some other directory, supply a complete pathname:

```
jim[1] /usr/ted/list
```

OR

jim[1] **../list**

Note: Only privileged users may create files in the **/cmd** directory. Even if you are privileged, you should check with the System Administrator before adding command files to **/cmd**.

As supplied, the Cromix-Plus system has some standard command files in **/cmd**. For example, the file named **/cmd/bak.cmd** deletes all files in the current directory with **bak** filename extensions. (For information about the Bak command, consult the Cromix-Plus User's Reference Manual.)

7.2 A Practical Use of the Path Command

Because the Shell looks first in the current directory for executable files (refer to chapter 4), you have complete freedom over the command files you create within your own directory structure. For example, if you have a file named **sort.cmd** (a personalized version of Sort) in your current directory, the following command executes *your* file--not the file **sort.bin** in the **/bin** directory:

jim[1] **sort infile > outfile**

To find out what file (if any) the Shell will execute in response to a particular command, use the Path command:

jim[1] **path sort**
/usr/you/sort.cmd

In this example, the Shell located the file in the current directory. Giving the Sort command now will execute the file **sort.cmd**.

Here, the Shell locates the file in the **/bin** directory:

jim[1] **path sort**
/bin/sort.bin

If you *do* have a file named **sort.cmd** it is not in the current directory. Giving the Sort command now will execute the file **sort.bin**.

To avoid confusion, it is a good idea to give your command files distinctive filenames, such as **mysort.cmd**. Once again, the Path command will tell you if the Shell can execute the file:

jim[1] **path mysort**
Command not found: "mysort"

"Command not found" means the file, if it exists, is not in any of the directories the Shell searches for executable files (refer to chapter 4). To execute the file, you must either change directories or supply a more complete pathname, as in the following examples:

```
jim[1] ls ../mysort
../mysort.cmd
jim[2] ../mysort infile > outline
jim[3]
```

The first command verifies the existence of the file in a user's home directory. The second command executes the file.

7.3 Redirection Within a Command File

Any command you can give from the command line can be put in a command file. This means you can use all the redirection techniques discussed in chapter 6.

Because command files can execute many commands, redirecting error messages is often desirable. You can even discard error messages (so they are neither displayed nor saved in a file) by redirecting them to a file named **null** in the /**dev** directory:

```
jim[1] del *.temp >* /dev/null
```

/dev/null is a null device. *Redirecting any kind of output to this file discards that output.*

Discarding error messages can be compared to ignoring road signs while driving on a highway. There is probably no harm in ignoring the sign

"Scenic route ahead"

but you would probably want to notice the sign

"Next gas station 300 miles ahead"

As there is no way to tell in advance how important error messages are going to be it is probably wise not to ignore them at all.

The next section introduces the Echo command, which is especially useful when writing command files. Using Echo, you can send informative messages to the terminal as a command file executes.

7.4 The Echo Command

Echo "echoes" its arguments to the terminal screen, as in the following example:

jim[1] **echo Hello there.**
Hello there.
jim[2]

You can echo "special" characters (such as an asterisk or pipe symbol) by quoting that character on the command line, as in the following example:

jim[1] **echo This is a special character:** ">" **right**
jim[2] This is a special character: > right

As an alternative, you may wish to quote the entire string of arguments, as in this example:

jim[1] **echo 'This is a special character: > right'**
This is a special character: > right

The Shell ignores special characters anywhere within single or double quotation marks. This is true when using Echo or any other utility. Without the quotation marks, the sample command redirects Echo's output to a file named **right**:

jim[1] **echo This is a special character: > right**
jim[2] **ty right**
This is a special character:
jim[3]

Chapter 3 contains a list of special characters you must quote if the Shell is to interpret them literally on a command line.

7.5 Command File Structure

Within a command file:

1. You can instruct the Shell to substitute arguments from the command line for up to nine numbered parameters in the command file.
2. You can specify labels and comments.
3. You can define string variables.
4. You can jump to labels using the Goto command, or do conditional jumps using Goto and If.

Items 1 and 2 are discussed more fully in the following subsections. After these subsections, the If and Goto commands are discussed.

Argument Substitution -- For #1 in a command file, the Shell substitutes the first argument from the command line, for #2, the second, and so on, through the ninth command-line argument (#9).

For example, the following command file displays two files--#2 (the second command argument) and #1 (the first command argument).

```
jim[1] ty display.cmd
ty #2
ty #1
```

The following command executes the file and displays the files **letter1** and **letter2**.

```
jim[1] display letter1 letter2
This is letter 2
.
.
.
This is letter 1
.
.
.
jim[2]
```

For #* in a command file, the Shell substitutes *all* command-line arguments:

```
jim[1] ty display_all.cmd
ty #*
jim[2] display_all letter1 letter2
This is letter 1
.
.
.
This is letter 2
.
.
.
jim[3]
```

For the asterisk, the Shell substitutes arguments in order--first, second, and so on, until there are no more arguments.

You can pass even more arguments to a command file using ambiguous file references on the command line. This example (again using the **display_all** command file) displays all the files in the

current directory with **.txt** filename extensions:

```
jim[1] display_all *.txt
This is text-file 1
.
.
.
This is text-file 2
.
.
.
This is text-file 3
.
.
.
```

Specifying Labels and Comments -- A percent sign anywhere on a line means the rest of the line is a comment. In the following example, "display all files" is a comment:

```
jim[1] ty display_all.cmd
ty #*                    % display all files
```

The Shell executes the command (ty #*) and ignores the comment. Comments are always ignored. Their only purpose is to make a command file's operation more understandable.

A comment at the beginning of a line--with no space after the percent sign--is a label. In the following example, "%start" and "%end" are labels:

```
jim[1] ty display_all.cmd
%start
ty #*                    % display all files
%end
```

As described in the next section, labels can affect the operation of a command file.

Shell Variables

It is possible to set and access string values stored in Shell variables. These include the standard Shell variables **#path**, **#ext**, **#abort** and **#err** as well as those defined by the user. Please refer to the discussions of "Shell" and "Set" in the *Cromix-Plus User's Reference Manual.*

7.6 The Goto Command

The Goto command transfers control within a command file. Goto's argument is always a line label.

To illustrate, consider this sample command file:

```
jim[1] ty echo_args.cmd
%start
echo #*
%end
```

All this file does is echo any command arguments.

Adding a Goto command causes the file to execute again and again.

```
jim[1] ty echo_args.cmd
%start
echo #*
goto start
%end
```

The command "goto start" starts the file executing once more from the beginning, the label **start**. Only CONTROL-C stops execution.

A Goto command always interrupts the sequential, line-by-line execution of a command file. After a Goto command, execution continues at another point in the command file, as determined by Goto's argument (a label). The process is commonly called "jumping" to a label.

Using Goto in combination with the If command (the next section), you can do "conditional" jumps. In other words, you can jump to a label if--and only if--some condition is satisfied. If the condition is not satisfied, the Goto command is ignored.

If you give the Goto command with a nonexistent line label, any commands in the file after Goto are not executed.

7.7 The If Command

The If command conditionally executes another command (often, the Goto command).

The most common forms of the If command are:

1. Execute the command if the previous command returned an error:

```
if -err command
```

All commands return a value to the Shell when they are through executing. The Shell, in turn, passes that value to the next command. A non-zero value indicates an error.

The "if -err" command tests for a non-zero return value. If the preceding command returns an error, the command following "if -err" executes, as in this example:

```
if -err goto end
```

If the preceding command returns a zero value, the command following "if -err" (in this case, Goto) does not execute.

There is additional information about return values in this chapter and in the Cromix-Plus User's Reference Manual.

2. Execute the command if a specified condition is true or false:

```
if string-1 = string-2 command
if string-1 != string-2 command
```

The relational operator (!=) means "not equal." Spaces must bracket both relational operators (= and !=).

For example, this line tests for an argument:

```
if #1hi = hi goto done
```

For #1, the Shell substitutes the first argument on the command line. In the absence of an argument, the Shell substitutes a null string (comparable to nothing) for #1. In this case, "#1hi" and "hi" are equal, and the Goto command is executed. If they are not equal (the condition specified by If is false), the Goto command is not executed.

Comparing #1 to a single character, such as a period (.), is the quickest way to test for an argument:

```
if #1. = .
```

You must compare #1 to *something* because the following command is syntactic nonsense:

```
if #1 =
```

You may be used to combining If with Else and Endif. Under the Cromix-Plus system, If-Else-Endif constructions are not possible. However, you can usually emulate their effects with consecutive If-Goto constructions.

7.8 The Shift Command

The Shift command shifts arguments in a command file. After a Shift command, #1 matches the *second* argument on the command line, #2, the *third*, and so on. Using Shift commands, you can cycle through a series of arguments.

If you want to return all command-file parameters to their original values, use the Rewind command.

7.9 The Rewind Command

The Rewind command cancels all preceding Shift commands in a command file. After a Rewind command #1 matches the first argument on the command line, #2, the second, and so on.

7.10 The Exit Command

Within a command file, Exit returns control to the Shell. Exit logs you out only when you give the command in response to the Shell prompt.

By giving the Exit command with an argument (a number), you can control the value Exit returns to the Shell. Any nonzero value will return an error. Without an argument, Exit returns whatever value it receives from the Shell.

The following command file, which sorts a series of files to the file **sortout**, contains several Exit commands. Clist displays the file, providing line numbers for reference (the line numbers and heading are not a part of the file):

```
jim[1] clist sort_all.cmd
File        SORT_ALL.CMD          Wednesday, November 14, 1984      10:14:29
     1      if #1. = . goto error     % if no arguments
     2      sort #* >> sortout        % sort input files
     3      exit
     4
     5      %error
     6      echo "Give me an argument!"
     7      exit 1
```

The first Exit command returns the value returned by the preceding command (Sort). The second Exit command returns an error value, regardless of the value returned by the preceding command (Echo).

Which Exit command executes depends on the outcome of the argument test (line 1). If true (#1. = .), execution jumps to **error**, and lines 6 and 7 execute.

```
jim[1] sort_all
Give me an argument!
jim[2]
```

If the argument test (line 1) is false (#1. != .), lines 2 and 3 execute. The rest of the file (lines 4 through 7) does not execute.

7.11 The Input and Strcmp Commands

The Input utility reads one line from STDIN and writes that line to STDOUT. The line is written when you press RETURN to end that line.

```
jim[1] input
This line is displayed when I press RETURN.
This line is displayed when I press RETURN.
jim[2]
```

By redirecting output, you can use Input to create a file containing one line of text.

```
jim[1] input > oneline
This line is written to a file when I press RETURN.
jim[2] ty oneline
This line is written to a file when I press RETURN.
jim[3]
```

Input, when used with Strcmp (the next command discussed), lets you communicate from the terminal with an executing command file.

7.12 The Strcmp Command

Strcmp tests for equality between STDIN and one or more text strings--ignoring letter case. For example, the following command tests the file **temp** for equality with the string **yes** or **please**.

```
ty temp | strcmp yes please
```

If the contents of the file is not identical to either of the two strings, Strcmp returns an error value. (Like all commands, Strcmp returns a value to the Shell when it is through executing.)

With the **-f** (for first) option, Strcmp tests the first character of STDIN for equality with the first character of one or more strings.

To test for a multiword string, enclose the string in quotation marks, as in this example:

```
ty temp | strcmp "yes please"
```

In response to this command, Strcmp returns an error if the contents of STDIN does not match the entire string "yes please."

Important: Do not conclude that Strcmp tests for a string embedded in the text of a file. *It does not.* Strcmp tests for equality between the file and the string. In other words, whether STDIN *contains* the string is irrelevant. The contents of STDIN must *be the same* as the string.

To test for embedded text strings, use the Match utility (refer to the Cromix-Plus User's Reference Manual).

The Strcmp utility is normally used in command files together with the Input utility. This command file runs the Shutdown utility if confirmation from the terminal is received:

```
jim[1] ty crash.cmd
echo "Do you want to shut down the system?"
input | strcmp -f y
if -err exit
shutdown
jim[2]
```

The first line echoes the phrase "Do you want to shut down the system?"

```
jim[3] crash
Do you want to shut down the system?
```

Input utility now waits for a line from the terminal. Any response (a single line) will be piped to the Strcmp utility that will compare the line received to the string "y". Suppose the user types yes and presses RETURN.

```
jim[4] crash
Do you want to shut down the system?
yes
```

The line containing the string "yes" is piped into the Strcmp utility which compares it with the string "y". The -f option causes the Strcmp utility to compare only the first character. The first characters are the same so that Strcmp return no error to the shell. As a result the exit command is not executed and the system shut down is initiated.

7.13 The Repeat Command

The Repeat command repeats a command a specified number of times, as in the following example:

```
jim[1] repeat 3 echo "this line is displayed three times"
this line is displayed three times
this line is displayed three times
this line is displayed three times
jim[2]
```

Other commands on a command line are not repeated:

```
jim[1] repeat 3 echo "Get to work!"; time
Get to work!
Get to work!
Get to work!
Monday, November 4, 1984                12:57:32
```

The semicolon (;) as a command separator is discussed in chapter 6.

In a command file, Repeat might be used to do multiple "shifts":

```
repeat 3 shift
```

After this command, the fourth command-line argument is equivalent to #1 in the command file.

7.14 The Scan Command

The Scan command is a sophisticated utility that can scan all files in a directory, including subdirectories, and do various things to each file. The full description of the Scan utility can be found in the *Cromix-Plus User's Reference Manual.* The following description gives just a few useful examples.

The general form of the command is:

scan **directory-pathname instruction-string**

Almost in all cases the instruction-string has to be in (single) quotes to prevent the Shell from trying to interpret it.

The simplest example is:

scan / **'print(path)'**

The scan utility will print the full pathname of every file in the file structure, just like the Ncheck utility described elsewhere.

The next example will print the pathnames of all files in the current directory (and its subdirectories)

that have the ".bak" extension:

scan . **'ext == ".bak" && print(path)'**

In a similar way you can get rid of all ".bak" files in your directory:

scan . **'ext == ".bak" && shell("del -v " | path)'**

The meaning is this: if the file has a ".bak" extension, execute a command. The command to be executed is

del -v filename

and this command has to be constructed by concatenating the string "del -v " and the filename supplied by the Scan utility. the "|" character (in this context) stands for concatenation.

A more sophisticated example is the same as above except that the user can first confirm (or not confirm) that the file is to be deleted. Note that as examples get more and more complicated they are more and more inconvenient to type. The Scan command is most useful when used in command files.

scan . **'ext==".bak" && print(path) && ok && shell("del "| path)'**

The logic behind this example is as follows:

For every file, if the extension is the right one,
then if the pathname can be printed (it always can),
then if the user confirms it,
then if it was successfully deleted
then do nothing.

If this command is run the Scan utility will print the pathname of every file with the extension ".bak" and wait for user confirmation. If the user types a "y" or a "Y" character, the file will be deleted.

7.15 Sample Command Files

This sample command file illustrates argument substitution, the Shift command, the Repeat command, and the Rewind command.

```
jim[1] ty shift_args.cmd
echo #*
shift
echo #1
repeat 2 shift
echo #1
rewind
echo #1
jim[2]
```

Executing the file first echoes all arguments. Then, after a shift, the second argument is echoed. After two more shifts, the fourth argument is echoed. A Rewind command cancels all shifts, and the first argument is again echoed.

```
jim[1] shift_args 1 2 3 4
1 2 3 4
2
4
1
jim[2]
```

The next example combines argument substitution with the Shift, If, and Goto commands:

```
jim[1] ty clist_all.cmd
%start
clist #1                % display file
shift
if #1. != . goto start
echo That's all!
```

To execute this file, you might give the following command:

```
jim[1] clist_all *.c
```

Before executing the file, the Shell generates the names of files in the current directory with **.c** (for C program) filename extensions. Then, the file executes as follows:

1. Clist displays the file corresponding to the first command argument (#1).
2. The Shift command shifts the arguments by one (#1 is now the second command argument, #2, the third, and so on).
3. The If command tests for another argument. If "#1." is *not* equal to ".", there is another argument--a filename generated by the Shell. In this case, the Goto command jumps to the label **start**.
4. The process repeats until "#1." (as determined by successive Shift commands) is equal to "."

By taking advantage of the Shell's ability to pass arguments to a command file, you can define options for the command files you write. For example, the following command executes the file **ccomp.cmd** with a user-defined option, **-l** (for link):

```
jim[1] ccomp -l *.c
```

"-l" is the first argument. The other arguments are the filenames in the current directory with **.c** (for C program) extensions.

Ccomp, written to compile and (optionally) link C programs, needs to know if "-l" is present so it can call Crolinker (the Cromemco linker) to link the programs to the Cromemco C libraries. During compilation and linking (if applicable), Ccomp must ignore "-l".

The next examples show portions of the file **ccomp.cmd**, illustrating how the Shift and Rewind commands solve the problem.

```
jim[1] ty ccomp.cmd
if #1.= -l.shift            % get rid of -l option
if #1. = . goto no-args     % if no other arguments
.
.
.
```

The Shift command shifts command-line arguments so "-l" (if present) is ignored. All arguments are now filenames, ensuring a successful compilation.

A Rewind command later in the file reinstates all arguments:

```
.
.
.
rewind
if #1. != -l. exit          % no linking requested
shift                       % get rid of -l option
crolinker #* usr/lib/clib /usr/lib/paslib
.
.
.
```

Then, Ccomp tests for "-l" once more. If linking is not required (#1. != -l.), the Exit command returns control to the Shell. If linking is required (#1. = -l.), the rest of the file executes:

1. The Shift command again gets rid of the option, so the next command (Crolinker) receives only filenames.
2. Crolinker then links all files to the Cromemco C-language libraries.

The file **ccomp.cmd** is shown in its entirety in the Cromix-Plus User's Reference Manual, under the discussion of the Exit command.

Appendix A - The Shell Command-Line Editor

The Shell command-line editor lets you correct or change a command line with a minimum of retyping. To illustrate, consider the following command line, in which the command name "scan" and the keyword "name" have been mistyped. The command-line editor is ideal for corrections like these.

```
System[1] scn / 'name == "repeat" && print(path)'
```

While you are typing the command, the LEFT and RIGHT ARROW keys move the cursor left and right on the command line--without erasing typed characters. Pressing the DELETE key deletes characters to the left of the cursor (one character is deleted each time you press the key).

Pressing CONTROL-I puts the editor in the Insert mode. With the editor in the Insert mode, any characters you type will be inserted to the left of the cursor. To take the editor out of the Insert mode, press ESCAPE. If the editor is not in the Insert mode, any characters you type replace existing characters.

When you are through editing a command line, pressing RETURN sends all *visible* characters to the Shell. The cursor's position when you press RETURN is therefore irrelevant.

A.1 Retrieving the Previous Command

One of the editor's most convenient features is its ability to retrieve, or "retype," a previous command.

You can easily demonstrate this feature by giving any command to the Shell. When the Shell displays a new prompt, press CONTROL-R, as in the following example:

```
jim[1] who
betty     qtty1     Nov-14-1984 07:16:22     3     1
jim       qtty2     Nov-14-1984 09:42:29     4     1
fred      qtty3     Nov-14-1984 09:53:47     5     1
% (CONTROL-R)
```

CONTROL-R redisplays your entire previous command.

```
jim[1] who
betty      qtty1      Nov-14-1984 07:16:22    3    1
jim        qtty2      Nov-14-1984 09:42:29    4    1
fred       qtty3      Nov-14-1984 09:53:47    5    1
jim[2] who
```

If you first type a number and then press CONTROL-R the command with that number, provided it exists, will be redisplayed.

It is also possible to move up and down through the list of previously entered commands using the UP and DOWN arrow keys. Additionally, if any characters appear to the left of the cursor prior to typing either arrow key, the editor will attempt to match a previous command with those characters. Commands retrieved in either manner may or may not be edited further and then executed by typing RETURN.

Cromemco®

280 Bernardo Ave.
P.O. Box 7400
Mountain View, CA 94039