IBM RT PC Advanced Interactive Executive Operating System Version 2.1

Using the AIX Operating System

Programming Family

Personal
Computer
Software

SC23-0794-0

IBM RT PC Advanced Interactive Executive Operating System Version 2.1

Using the AIX Operating System

Programming Family

Personal
Computer
Software

First Edition (January 1987)

Portions of the code and documentation described in this book were developed at the Electrical Engineering and Computer Sciences Department at the Berkeley Campus of the University of California under the auspices of the Regents of the University of California.

This edition applies to Version 2.1 of IBM RT PC AIX Operating System Licensed Program, and to all subsequent releases until otherwise indicated in new editions or technical newsletters. Changes are made periodically to the information herein; these changes will be reported in technical newsletters or in new editions of this publication.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent program may be used instead.

International Business Machines Corporation provides this manual "as is," without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this manual at any time.

Products are not stocked at the address given below. Requests for copies of this product and for technical information about the system should be made to your authorized IBM RT PC dealer or your IBM marketing representative.

A reader's comment form is provided at the back of this publication. If the form has been removed, address comments to IBM Corporation, Department 997, 11400 Burnet Road, Austin, Texas 78758. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

About This Book

A computer system has two main parts:

Hardware The physical components of the system.

Software The programs that control how the hardware works.

On the IBM RT Personal Computer[1] system, there are two types of software:

The *operating system*
: A set of programs that controls how the system works. Some of the tasks an operating system usually performs are allocating system resources, scheduling operations, and controlling the flow of data through the system.

Application programs
: Software that performs a particular task such as word processing, project planning, or inventory control.

The AIX[1] Operating System consists of a ***kernel*** (the programs that control how the system works) and a ***shell*** or (***command interpreter***). The shell provides a set of ***commands*** (programs) that cause the system to perform specific operations. The subject of this book is the AIX[1] Operating System —a part of the RT PC[1] system that exists between the hardware and the application programs. To determine whether you want to work with the AIX Operating System as this book describes, please read the remainder of this section.

---

1 RT, RT PC, RT Personal Computer, and AIX are trademarks of International Business Machine Corporation.

The AIX Operating System is based on UNIX[2] System V, which consists of a kernel and a standard UNIX shell (the Bourne shell). Your RT PC system may have additional command interpreters installed, for example:

- **csh** (described under **csh** in *AIX Operating System Commands Reference*
- **DOS Services** (described in *Using AIX Operating System DOS Services*)
- **Usability Services** (described in *Usability Services Guide*).

This book includes the information you need to be able to:

- Start your RT PC system and use simple commands
- Display and print the contents of files
- Use the AIX file system
- Work with processes
- Write shell programs.

Who Should Read This Book

This book is written for those who want to learn how to use the basic features of the AIX Operating System.

---

2 UNIX was developed and licensed by AT&T. It is a registered trademark of AT&T in the United States of America and other countries.

Before You Begin

Before you can begin to work with this book, your RT PC system must be set up and the AIX Operating System must be installed. In addition, you should have a **user name** and possibly a password. If your system is not installed, see *Installing and Customizing the AIX Operating System.* If you need a user name, see *Managing the AIX Operating System.*

Note: Beginning with Chapter 2, many of the tasks in this book require you to use one of the RT PC text editing programs. When you begin Chapter 2, please pay careful attention to the list of text editing programs in "About This Chapter" on page 1-3.

How to Use This Book

This book also is divided into chapters, each devoted to a major AIX Operating System concept or feature:

- Chapter 1, "Getting Started on the AIX Operating System" on page 1-1, explains how you gain access to the system and how you use AIX Operating System commands to perform work on the system. After you are familiar with the information in this chapter, you should be able to use your system to follow the examples in the remainder of the book.

- Much of the work you do on the system produces or modifies ***files*** (collections of data stored together in the computer under one name). Chapter 2, "Displaying and Printing Files" on page 2-1 explains some of the ways in which you can view the contents of files, either by displaying them on the screen or by printing them.

- Chapter 3, "Using the File System" on page 3-1 explains how the AIX ***file system*** works, one of the most important AIX Operating System concepts. Besides explaining the structure and principles of the file system, this chapter also has many examples of file system use that you can follow on your system.

- At any given time, when your system is running, there are a number of different ***processes***, each performing a different task.

Chapter 4, "Understanding Processes" on page 4-1 explains the relationship between programs and processes and some basic ways to run processes, modify the way processes work, and monitor their progress.

- Chapter 5, "Using the Shell with Processes" on page 5-1 explains some of the more advanced ways to control processes, using features of the AIX Operating System shell program.

- Chapter 6, "Using Advanced Shell Features—A Reference" on page 6-1 contains information about the shell that you may find useful if you are developing your own shell programs or procedures.

In addition, this book also includes supplemental information in an appendix, and a glossary and an index to make it easier for you to find information.

A Reader's Comment Form and Book Evaluation Form are provided at the back of this book. Use the Reader's Comment Form at any time to give IBM information that may improve the book. After you become familiar with the book, use the Book Evaluation Form to give IBM specific feedback about the book.

You can use this book in one of two ways:

- As a training manual. Read and work through it from the first chapter to the last one. This should give you a general understanding of the AIX Operating System.

- As a reference manual. Use the "Contents" and "Index" to locate particular topics. This is a good way to refresh your memory or learn more details about the AIX Operating System.

Special Features

This book uses type style to distinguish among kinds of information. General information is printed in the standard type style (the type style used for this sentence). The following type styles indicate other types of information:

New terms
Each time a new term is introduced, its first occurrence is printed in this type style (for example, "the AIX Operating System ***file system***").

System parts
The names for keys, commands, files, and other parts of the system are printed in this type style (for example, "the **cp** command").

Variable information
The names for information that you must provide are printed in this type style (for example, "type *yourname*").

`Special characters`
Any characters that have a special meaning are printed in this type style (for example, "the & and && operators have different uses"). This type is also used for the names of files that you create as you work through this book (for example, "create a file named `afile`").

`Information you are to type`
Many examples in this book are designed for you to try them on your own system; the information that you should type is printed in this type style (for example, "type `ls text` and press **Enter**").

Where appropriate, the chapters and major sections of this book begin with a box containing quick reference material, for example:.

To Use a Quick Reference Box

1. Skip the quick reference boxes the first time you read a section.
2. Use the quick reference boxes as a fast path through the book.
3. Refer to the quick reference boxes to refresh your memory.

You should use the boxes for reference after you are generally familiar with the contents of a section or chapter. You can skip the box the first time you read a section. The boxes make a convenient ***fast path*** through the book, but they are not comprehensive and they are not intended to take the place of the explanatory material in each section.

After the quick reference boxes, each chapter in this book takes the same general approach to the topics it covers—a series of explanations and examples. The examples build upon each other; in many instances, an example uses a file created in a previous example. Therefore, if you intend to follow the examples on your system, it is important for you to work through each chapter from start to finish.

In the examples, the characters you should type are printed in blue, as this example shows:

```
$ ls
  afile
  bfile
  cfile
$ _
```

After you type the characters on a line, press the **Enter** key.

In the text, whenever you are told to ***enter*** a command or other information, you should type the information and then press the **Enter** key.

Also included in the binder with this book are two aids to using the AIX system:

- A Quick Reference Card, which contains the essential steps for a number of basic tasks. The Quick Reference Card should be most useful after you are generally familiar with the information in the book.

- A keyboard reference chart for four of the different display stations that you can use with the AIX system. Certain special functions require a different sequence of keys on different keyboards. The keyboard reference chart for a particular display station shows you which keys to press on that keyboard to produce the special function.

Related Books

- *IBM RT PC Installing and Customizing the AIX Operating System* provides step-by-step instructions for installing and customizing the AIX Operating System, including how to add or delete devices from the system and how to define device characteristics. This book also explains how to create, delete, or change AIX and non-AIX minidisks.

- *IBM RT PC Managing the AIX Operating System* provides instructions for performing such system management tasks as adding and deleting user IDs, creating and mounting file systems, and repairing file system damage.

- *IBM RT PC AIX Operating System Commands Reference* lists and describes the AIX Operating System commands.

- *IBM RT PC Guide to Operations* describes the IBM 6151 and IBM 6150 system units, the displays, keyboard, and other devices that can be attached. This guide also includes procedures for operating the hardware and moving the IBM 6151 and IBM 6150 system units.

- *IBM RT PC Problem Determination Guide* provides instructions for running diagnostic routines to locate and identify hardware problems. A problem determination guide for software and

three high-capacity (1.2MB) diskettes containing the IBM RT PC diagnostic routines are included.

- *IBM RT PC Usability Services Guide* shows how to create and print text files, work with directories, start application programs, and do other basic tasks with Usability Services. (Packaged with *Usability Services Reference*)

- *IBM RT PC Usability Services Reference* supplements *IBM RT PC Usability Services Guide* by including information on using all of the Usability Services commands. (Packaged with *Usability Services Guide*)

- *IBM RT PC Exploring Usability Services* is an online tutorial for first-time users of the Usability Services. This tutorial simulates the user interface and shows how to use the keyboard and the optional mouse, how to manipulate windows, and how to use files and directories.

- *IBM RT PC Messages Reference* lists messages displayed by the IBM RT PC and explains how to respond to the messages.

- *IBM RT PC Using AIX Operating System DOS Services* provides step-by-step information for using AIX Operating System shell. (Available optionally; packaged with *IBM RT PC AIX Operating System DOS Services Reference*)

- *IBM RT PC AIX Operating System Technical Reference* describes the system calls and subroutines that a C programmer uses to write programs for the AIX Operating System. This book also includes information about the AIX file system, special files, file formats, GSL subroutines, and writing device drivers. (Available optionally)

- *IBM RT PC AIX Operating System Communications Guide* provides information and customizing details on communicating with other users in a multiple work station environment and in two system-to-system configurations. One configuration links a single interactive work station with a remote host, and the other links two AIX-based systems for batch file transfer.

- *IBM RT PC INed* provides guide and reference information for using the INed program to create and revise files.

- *IBM RT PC AIX Operating System Programming Tools and Interfaces* describes the programming environment of the AIX Operating System and includes information about using the operating system tools to develop, compile, and debug programs. In addition, this book describes the operating system services and how to take advantage of them in a program. This book also includes a diskette that includes programming examples, written in C language, to illustrate using system calls and subroutines in short, working programs. (Available optionally)

Ordering Additional Copies of This Book

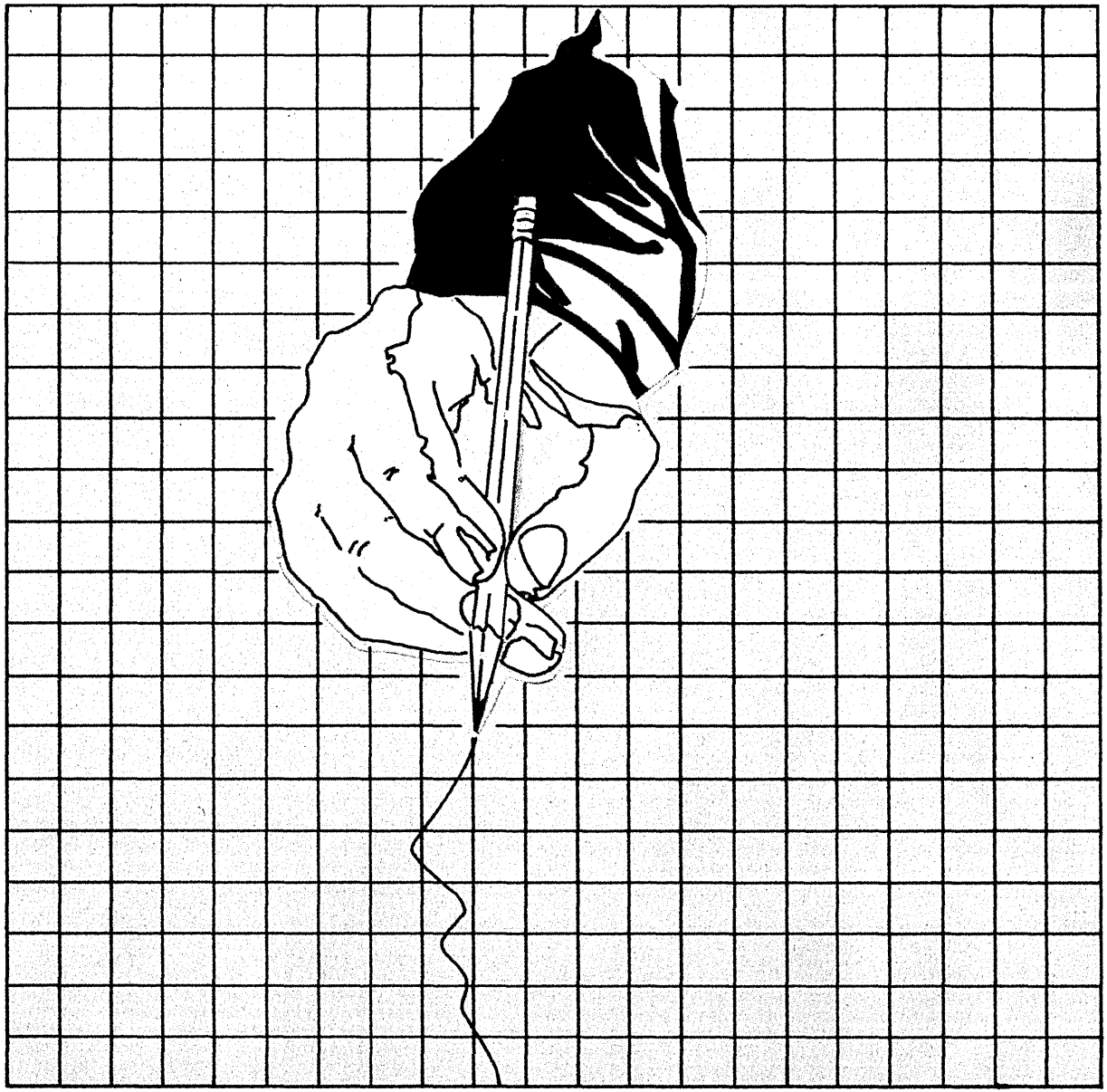
To order additional copies of this publication (without program diskettes), use either of the following sources:

- To order from your IBM representative, use Order Number SBOF-0169.
- To order from your IBM dealer, use Part Number 92X1269.

A binder, the *Using the AIX Operating System* manual, the *AIX Operating System Quick Reference Card*, and the keyboard reference chart set are included with the order. For information on ordering the binder, manual, and quick reference separately, contact your IBM representative or your IBM dealer.

Contents

Chapter 1. Getting Started on the AIX Operating System



CONTENTS

About This Chapter

This chapter introduces you to the basic tasks of using the RT PC AIX Operating System. If you are not familiar with the components of the system, see *Guide to Operations* to identify which components your system has and where their power switches are.

After you finish this chapter, you should learn how to create and modify files with a ***text editing program.*** (A ***file*** is simply a collection of data stored together under a given name.) The following editing programs are available on the AIX system:

- **ed** (see Appendix A, "Creating and Editing Files with **ed**" on page A-1)
- **INed**[1] (see *INed*)
- **vi** (see *AIX Operating System Commands Reference*).

Your system may have other editing programs as well.

Once you complete this chapter and learn how to use an editing program, you should have the basic skills necessary to start using the operating system.

Note: Before you can work through this book on your AIX system, the components of your system must be set up and the AIX Operating System must be installed. In addition, you may be required to have a user name and a password. If your system is not set up and installed, see *Installing and Customizing the AIX Operating System.* If you do not have a user name and a password, see *Managing the AIX Operating System.*

1 INed is a registered tradmark of INTERACTIVE Systems Corporation.

Logging In to the AIX Operating System

Before you can use the AIX Operating System, your system must be running and you must be ***logged in*** (identified as a valid system user). If your system displays the `login` prompt after going through its initial operations, log in with the procedure described in the following quick reference box. If your system displays the `autologin of` *name* prompt, it has logged you in automatically; you can continue your work in this chapter at "Logging Out of the AIX Operating System" on page 1-6.

To Log In to the AIX Operating System

If your system is not running:

1. Turn on the power switch for each component.
2. Continue with step 3.

If your system is already running:

3. After the prompt:

   `login:`

   enter your user name.

4. After the prompt:

   `password:`

   enter your password.

   (If you do not have a password, you do not receive the `password` prompt.)

Starting the System

When you start the RT PC system, the system goes through a series of internal procedures before it is ready to use. When the system is ready it displays a copyright notice and the following prompt:

```
login:
```

Logging in to the System

When the system displays the `login` prompt, type your user name and press **Enter**. (If your system displays the `autologin of` *name* prompt, it has logged you in automatically.)

After you enter your user name, the system displays the `password:` prompt. Enter your password. For security reasons, the system does not display your password as you type it. (If you do not have a password, the system does not display the `password` prompt.)

Note: Although your system may not require you to have a password, it usually is good practice to set a password for yourself anyway. For an explanation of how to set or change your password, see "Setting and Changing Your Password" on page 1-10.

When the system completes the login procedure, it displays a prompt, usually: `$` (a dollar sign followed by a space, which is called the ***shell prompt***). The system is ready to accept a command.

Note: The usual prompt is `$` (the shell prompt). Another standard prompt is `#`. However, it is possible for the prompt to be set to some other character or characters.

If your system does not display a prompt, you are not logged in. Try to log in again. You may have typed your password incorrectly. If you still cannot log in, see *Managing the AIX Operating System.*

Logging Out of the AIX Operating System

To Log Out and Stop the Operating System

The $ (shell) prompt must be displayed before you can log out of the system correctly.

To log out and leave the operating system running:

- Press **END OF FILE (Ctrl-D)**.

To stop the operating system in the multi-user mode:

- Enter `shutdown`

To stop the operating system in the single user mode:

- Enter `shutdown -f`

If you simply want to log out of the system, but leave the system running for other users, press **Ctrl-D**. After logging you out, the system displays the `login` prompt for the next user.

If you want to turn off the power to the system, you must first stop the operating system in an orderly way with the **shutdown** command.

Warning: It is very important that you use the **shutdown** command before you turn off the power to your system. Failure to do so may result in the loss of data.

When the operating system stops running, you receive the message:

```
....Shutdown completed....
```

Note: On some systems, only selected users can use the **shutdown** command. If you are not responsible for shutting down your system, simply log out and leave the system running.

For more information about **shutdown**, see *Managing the AIX Operating System.*

Using Operating System Commands

Operating system ***commands*** are programs that perform tasks on the AIX system. The AIX Operating System has a large set of commands which are described in the remaining chapters of this book and in *AIX Operating System Commands Reference.* In addition to the commands provided with the system, it is possible for you to create your own commands (see "Writing and Running Shell Procedures" on page 5-16).

When you work with the operating system, you typically enter commands after the $ (shell) prompt on the ***command line***, for example:

```
$ ls
```

The **ls** (list) command displays the contents, if any, of your ***login directory***.

Note: If you make a mistake while typing a command, use the **Backspace** key to erase the incorrect characters and then retype them. The ← cursor movement key does not work for this purpose.

A ***flag*** alters the way a command works. Most commands have several flags. For example, the **l** (long) flag to the **ls** command provides more information about the contents of the directory. The following example shows how to use the **-l** flag with the **ls** command:

```
$ ls -l
```

An ***argument*** is a string of characters, usually the name of a file or directory, that follows a command name. An argument specifies what data the command is to work with. If you use flags with a command, arguments follow the flags on the command line. In the following example, **/bin** (the name of a directory) is an argument:

```
$ ls -l /bin
```

The **ls -l /bin** command gives a long listing of the contents of the directory **/bin**.

Note: Chapter 3, "Using the File System" on page 3-1 contains a detailed explanation of files and directories.

If you start a command and then decide that you do not want it to complete, press **Ctrl-Backspace** or **(Left)Alt**-Pause (**INTERRUPT**) to cancel it. (**INTERRUPT** and other special function keys are described under "Using Special Functions and Keyboard Reference Charts" on page 1-15.)

Note: While a command runs, the system does not display the $ (shell) prompt. When the command completes, the system displays the $ prompt again, indicating that you can enter another command.

Setting and Changing Your Password

A user name is a code that you use to identify yourself to the system. A ***password*** is a code that you use to verify your identity. Unlike your user name, which is public information and does not usually change, your password is private and you should change it periodically (with the **passwd** command) to protect your data from unauthorized access. If your account does not have a password, use the **passwd** command to set one.

To Set or Change Your Password

1. Enter:

   ```
   passwd
   ```

2. After the prompt:

   ```
   Old password:
   ```

   enter your old password.

3. After the prompt:

   ```
   New password:
   ```

   enter your new password.

4. After the prompt:

   ```
   Re-enter new password:
   ```

   enter your new password again.

Using the passwd (Change Password) Command

Your password must be at least four characters long and can consist of letters, numbers, and punctuation marks. If your password contains only one type of character (for example, lowercase letters), it must be at least six characters long. The system recognizes only the first eight characters in a password.

On most systems, you can change your password as often, or rarely, as you like. However, passwords can be set with limits on how frequently they can be changed or how long they remain valid. (For information about setting password time limits, see *Managing the AIX Operating System.*)

To set or change your password, enter the `passwd` command:

```
$ passwd
Changing password for username
Old password:
```

After the `Old password` prompt, enter your old password. (If you do not have an old password, you do not receive the prompt.)

Note: For security reasons, the system does not display your password as you type it.

After the system verifies your old password, it is ready to accept your new password:

```
New password:
Re-enter new password:
$ _
```

Finally, to verify the new password (since you cannot see it as you type), the system prompts you to enter the new password again. When the $ (shell) prompt returns to the screen, your new password is in effect.

Note: You should not forget your password. You cannot log in to the system without it. However, if you do forget your password, see *Managing the AIX Operating System* to learn how to remove

password protection from your account. If this happens to you, promptly set a new password that you can remember.

Using Display Station Features

You can accomplish most of the tasks described in this book with a very few special keys in addition to the typewriter-style letter and number keys. Figure 1-1 on page 1-14 shows the standard IBM RT PC Keyboard:

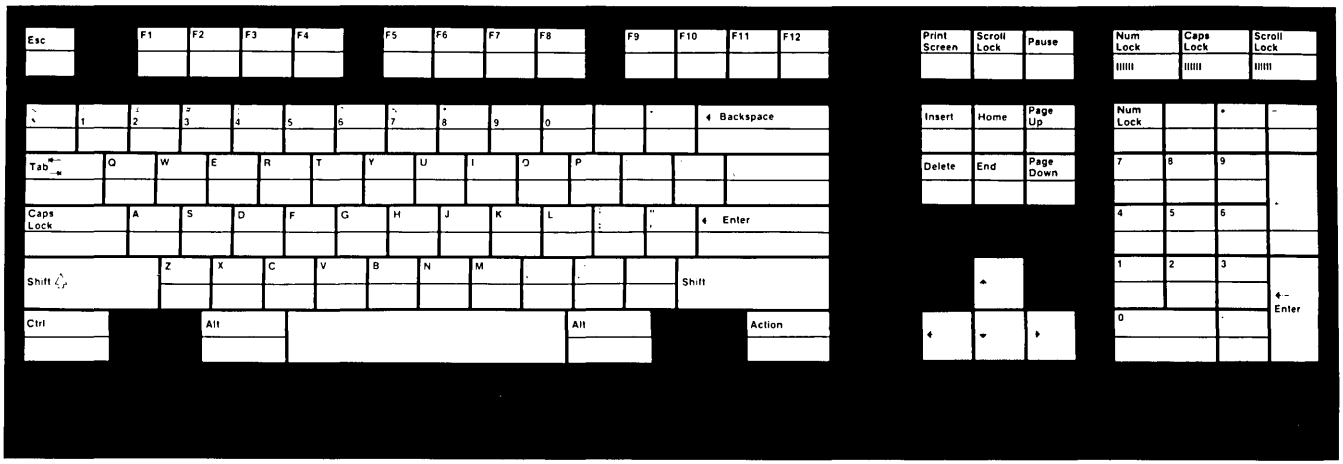


Figure 1-1. The IBM RT PC Keyboard

(Left)Alt Used with other keys for special functions. (There are two keys labeled **Alt**. **(Left)Alt** specifies the **Alt** key on the left side of the keyboard. The **Alt** key on the right side of the keyboard does not perform the same function.)

Ctrl	Used with other keys for special functions.
Enter	Sends from the keyboard to the system and moves the cursor from the end of one line to the beginning of the next.
Esc	Ends certain system activities and is used with other keys for special functions.
←	(**Cursor Left**) Moves the cursor to the left.
→	(**Cursor Right**) Moves the cursor to the right.
↑	(**Cursor Up**) Moves the cursor up.
↓	(**Cursor Down**) Moves the cursor down.

The ***cursor*** is the underscore or rectangle on your screen that moves as you type characters or press the cursor movement keys.

Using Special Functions and Keyboard Reference Charts

To perform certain functions, you must use two or more keys together. For example, to log out of the AIX Operating System, you must send the **END OF FILE** (end of file) signal. To do so, you press and hold the **Ctrl** key and then press the **D** key. The names of special functions are printed in this book in all uppercase letters. Following is a list of the special functions and the IBM RT PC Keyboard keys you press to use them:

DUMP	**Ctrl-(Left)Alt-End** (kernel dump)
END OF FILE	**Ctrl-D**
INTERRUPT	**(Left)Alt-Pause**
NEXT WINDOW	**Alt-Action**
QUIT WITH DUMP	**Ctrl-V** (application dump)

SOFT IPL	**Ctrl-(Left)Alt-Pause**
RESUME OUTPUT	**Ctrl-Q**
STOP OUTPUT	**Ctrl-S**
HORIZONTAL TAB	**Ctrl-I**
VERTICAL TAB	**Ctrl-K**
FORM FEED	**Ctrl-L**
CARRIAGE RETURN	**Ctrl-M** (same function as **Enter**)
LINE FEED	**Ctrl-J**

Ctrl-J works sometimes when the system will not process **Enter**. If you get strange information on the screen or the system does not respond when you press **Enter**, reset the characteristics of the display station with the following command:

Ctrl-J `stty sane echo -tabs` **Ctrl-J**

You can use the AIX Operating System from any of several different display stations, each of which has a different keyboard:

- The main RT PC display station, sometimes called the ***console*** (which includes the IBM RT PC Keyboard)
- The IBM 3161 ASCII Display
- The DEC VT100[1]
- The DEC VT220[2]

---

1 DEC and VT100 are registered trademarks of Digital Equipment Corporation.

2 DEC and VT220 are registered trademarks of Digital Equipment Corporation.

- The IBM PC, using the Crosstalk XVI[3] program.

At the back of this book is a keyboard reference chart for each of these keyboards. Use the keyboard reference chart for your keyboard to determine which keys on that keyboard produce the special functions.

---

3 Crosstalk XVI is a registered trademark of Microstuf Company, Inc.

Setting Display Station Characteristics

You can use the **stty** command to modify how a display station works. The general format of **stty** is:

`stty` *flag*

Following is a list of the **stty** flags you may find useful while you learn to use the system:

-a Displays the current **stty** settings.

echoe Causes the **Backspace** key to erase characters when you backspace over them.

enhedit Makes special features available for editing the command line. The **-enhedit** flag makes the special features not available. For more information on the special command line editing features, see "Special Command Line Editing Features" on page 1-20.

page Causes the system to display information one screen at a time (instead of scrolling through the entire output of a command without pause). When **page** is set, press another key (for example, **Enter**) to display the next screen of information. To disable the **page** setting, use the **-page** flag. **-page** is the normal setting. Use **page** in conjunction with **length** to set the number of lines displayed.

length *n* Sets screen length to *n* lines, where *n* is a number from 1 to 255. **length** works only in conjunction with **page**.

In the following example, the **stty** command enables the special command line editing features, sets the system to display information one screen at a time, and sets the length of the screen to 22 lines:

```
$ stty enhedit page length 22
$ _
```

For more information about **stty** flags, see **stty** in *AIX Operating System Commands Reference*. For information about running **stty** automatically and additional display station features, see *Managing the AIX Operating System*.

Special Command Line Editing Features

The **enhedit** flag of the **stty** command provides certain command line editing features that you may find convenient. Each time you enter a command, the system stores a copy of the command line in a temporary storage area called a ***buffer***. The buffer holds up to eight lines, or ***templates***. You can change the buffer size with the **enhedstack** parameter in **/etc/master**. As you move the cursor on the command line, a ***marker*** moves to the equivalent position in the buffer. Use the ↑ (cursor up) and ↓ (cursor down) keys to change to another template in the buffer. When you have changed to the last template in the buffer, the buffer wraps around to the first template. Using the information stored in the buffer, the command line editing features can replace all or part of the last eight lines you entered from the terminal, making it easy to enter the same line again or modify a previous template to produce a similar line. To enable this set of features (described in the following list), enter `stty enhedit` on a local terminal or `stty enhedit ascedit` on a remote terminals:

Note: The following list uses key names as they appear on the IBM RT PC Keyboard. If you are using a different keyboard, see the appropriate keyboard reference chart. The keys for remote terminals are shown in parentheses.

Key	Name and Function
F1 or → **(Esc 1)**	**Display character** Either of these keys displays one character from the current template each time you press it.
F2 **(Esc 2)**	**Display before** Displays all characters of the current template line from the buffer before the first occurrence of the next character you type (but after the position of the marker in the buffer).

F3
(Esc 3) **Display line**

Displays all of the characters of the current template line.

F4
(Esc 4) **Display after**

Moves the marker to the first occurrence of the next character you type, but does not display any characters. Use F1, F2, or F3 to display characters after positioning the marker with F4.

F5
(Esc 5) **Load buffer**

Places the contents of the input line into the buffer replacing the current template (without running the command).

Backspace
or ← **Erase character**

Either of these keys erases the preceding character each time you press it, allowing you to correct typing errors on an input line.

Note: If you are not using the special editing features, you can use the **Backspace** key to correct errors, but not the ←, which erases characters from the input line, but not from the buffer template.

Insert
(Esc i) **Insert character**

Turns the insert mode on or off. When insert mode is on, you can insert characters between other characters on the line without changing the position of the marker in the buffer.

Delete
(Esc d) **Skip character**

Moves the marker in the buffer one position to the right without erasing the skipped character. The cursor remains in the same position on the input line.

Esc
(Esc Esc) **(Ctrl-U) Erase command line**

Erases the entire input line, but does not affect the contents of the buffer.

↑
(Esc h) **Display next command line**

Moves the marker to the next line template in the buffer, and displays the entire template.

↓
(Esc l) **Display previous command line**

Moves the marker to the previous line template in the buffer, and displays the entire template.

Using Virtual Terminals

It is easier to understand the concept of virtual terminals if you understand something about how the AIX system handles commands. Ordinarily, you enter a command, wait for the command to complete, and then enter your next command. However, the AIX system can run more than one command at the same time. For example, if your system has more than one display station, you can enter a command at one display station, move to the next display station, enter another command, and so on, until you have commands running at every display station on the system.

The RT PC ***virtual terminal*** feature gives you the equivalent of multiple display stations with one exception: Rather than moving from one display station to the next to enter different commands, you stay at the main display station and use commands and keys to move from one virtual terminal to the next.

Note: Only the main RT PC display station has the virtual terminal feature. If you try to use virtual terminals on the main display station and find that they do not work, see *Managing the AIX Operating System* for an explanation of how to make the virtual terminal feature available.

To Use Virtual Terminals

1. To start a virtual terminal, enter:

   ```
   open sh
   ```

2. To move from one virtual terminal to another, press:

   NEXT WINDOW (Alt-Action)

3. To close (stop) a virtual terminal, press:

   END OF FILE (Ctrl-D)

4. Before you log out, close all virtual terminals you have opened.

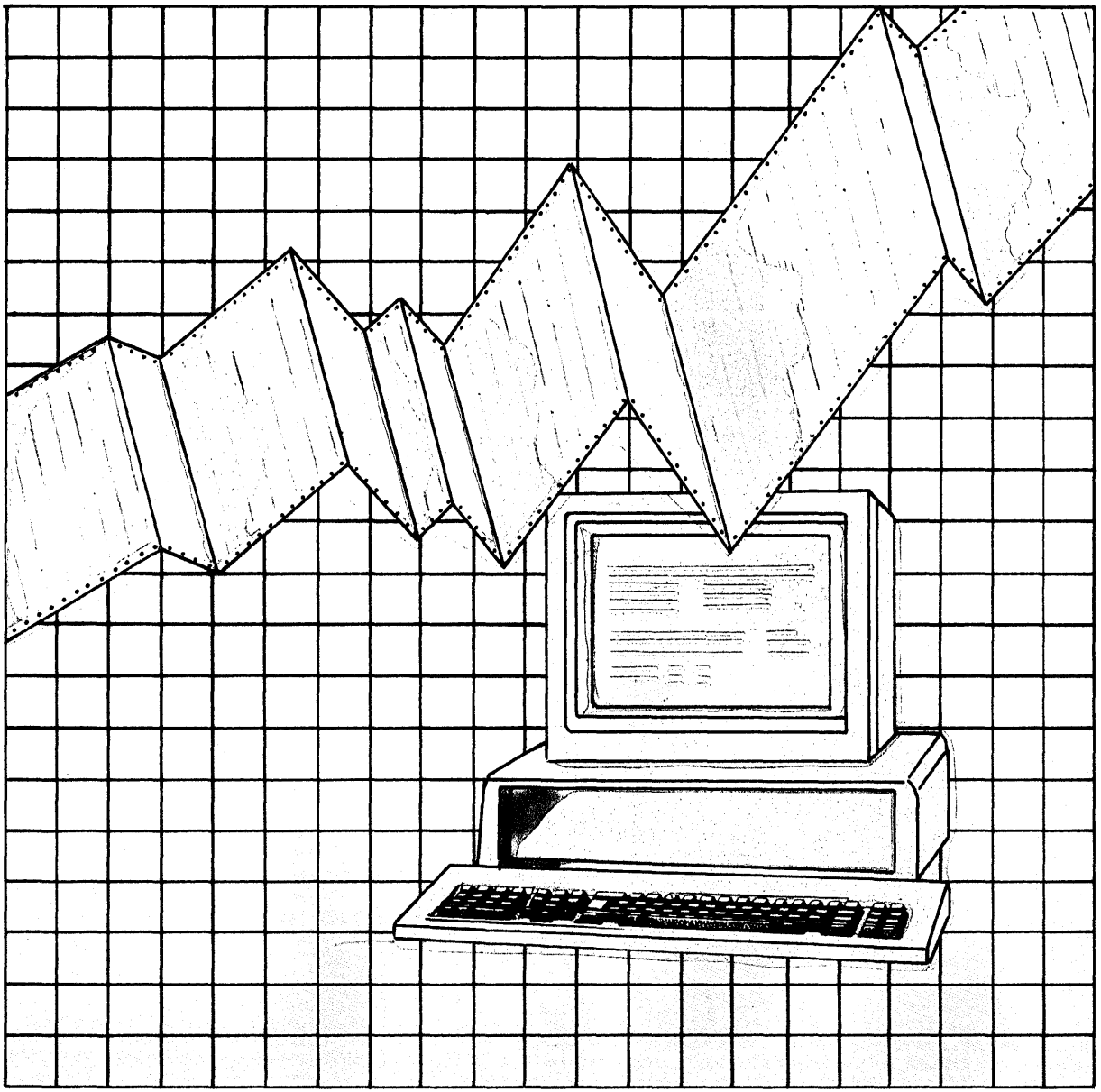
Note: The **open sh** command opens a virtual terminal with the standard operating system command interpreter (**sh**, or the shell). If you want to open a virtual terminal with a different command interpreter, substitute the name of that program for **sh**.

After you open a virtual terminal, you can enter a command just as you normally would. If you have several virtual terminals open, **NEXT WINDOW** moves you from one to the next, in the order in which you opened them, as though they are connected in a ring. The maximum number of virtual terminals that the system can have open concurrently is 16.

For more information about virtual terminals and other features of the main display station, see *Managing the AIX Operating System.*

Note: Before you continue with the remainder of this book, you should become familiar with a text editing program. See "About This Chapter" on page 1-3 for a list of the editing programs available on the AIX system.

Chapter 2. Displaying and Printing Files



CONTENTS

About This Chapter

When you want to see the contents of a file, you have two options:

- Display the file on the screen.
- Print the file on the system printer.

In either case, you can choose to see the file:

Unformatted Just as it is.

Formatted With particular characteristics such as double spacing, a heading for each page, the number of characters per line, and the number of lines per page.

This chapter explains how to display and print your files, with or without formatting.

Note: This chapter requires you to work with three files that you create with a ***text editing program***. "Creating Sample Files for This Chapter" on page 2-4 explains how to create these files. Later chapters of this guide, however, require you to be generally familiar with one of the AIX text editing programs, which are listed in "About This Chapter" on page 1-3.

Creating Sample Files for This Chapter

Before you can work through the examples in this chapter, you must have three files to work with. The following example shows how to use the **ed** program to create the files used in this chapter. If you are familiar with a different editing program, you can use that program to create these files. If you already have created three files with an editing program, you can use those files by substituting their names for the file names used in the examples.

```
$ ed file1
?file1
a
You start the ed program by entering
the command ed, followed by the name
of a new or existing file.
.
w
101
e file2
?file2
a
If you are creating a new file, the ed
program first displays the prompt ?.
You then type an a on a line by itself to
indicate that you are ready to add text to the
file.
.
w
171
e file3
?file3
a
When you finish entering your text, enter a period
on a line by itself.  Then enter w to write (or save)
a copy of the new file.
.
w
129
q
$ _
```

Displaying Files—The pg (page) Command

If the $ (shell) prompt is on your screen, you can use the **pg** (page) command to display the contents of one or more files. You also can use an editing program to display files.

To Display a File

Enter:

`pg` *filename*

Where *filename* can be a file name or a series of file names separated by spaces.

Displaying Files Without Formatting—The pg (page) Command

In the following example, the `pg` command displays the contents of the file named `file1`:

```
$ pg file1
You start the ed program by entering
the command ed, followed by the name
of a new or existing file.
$ _
```

You can use the **pg** command to display the contents of more than one file at a time:

```
$ pg file1 file2
You start the ed program by entering
the command ed, followed by the name
of a new or existing file.
If you are creating a new file, the ed
program first displays the prompt ?.
You then type an a on a line by itself to
indicate that you are ready to add text to the
file.
$ _
```

The pg command displays the contents of file1 first, then the contents of file2, with no break between them. The **pg** command always displays files in the order in which you list them (left to right).

When you display files that contain more lines than will fit on the screen, the **pg** command pauses as it displays each screen. To see the next screen of information, press **Enter**.

Formatting Files for Display—The pr Command

Formatting is the process of controlling how the contents of your files look when they are displayed or printed. The **pr** command does simple file formatting. Used without any flags, the **pr** command does the following:

- Divides the contents of the file into pages
- Puts the date, time, page number, and file name in a heading at the top of each page
- Leaves five blank lines at the end of the page to skip over the perforations between sheets of continuous form paper.

Flags to the **pr** command give you considerable control over how your files format. Figure 2-1 on page 2-7 explains several of these flags. All of the **pr** command flags are covered under **pr** in *AIX Operating System Commands Reference.*

To Format a File with The pr Command

Enter:

pr *filename*

When you use the **pr** command to format a file for display, the contents of the file may scroll up and off of your screen too quickly to be read. Use the **stty page** command (described in "Setting Display Station Characteristics" on page 1-18) to cause the system

to pause at the end of each page; press **Enter** to display the next page.

A number of flags to the **pr** command let you control the way **pr** formats your files. Figure 2-1 explains some of the most useful **pr** command flags. (These and other flags also are covered under **pr** in *AIX Operating System Commands Reference.*)

Flag	Action	Example
+*num*	Begins formatting on page number *num*. Otherwise, formatting begins on page 1.	`pr +2 file1`
-*num*	Formats page into *num* columns. Otherwise, **pr** formats pages with one column.	`pr -2 file1`
-m	Formats all specified files at the same time, side-by-side, one per column.	`pr -m file1 file2`
-d	Formats double-spaced output. Otherwise, output is single spaced.	`pr -d file1`
-w*num*	Sets line width to *num* characters. Otherwise, line width is 72 characters.	`pr -w40 file1`
-o*num*	Offsets (indents) each line by *num* character positions. Otherwise, offset is 0 character positions.	`pr -o5 file1`
-l*num*	Sets page length to *num* lines. Otherwise, page length is 66 lines.	`pr -l30 file1`

Figure 2-1 (Part 1 of 2). pr Command Flags

Flag	Action	Example
-h	Uses next string of characters, rather than the file name, in the header. If the string includes blanks or special characters, it must be enclosed in quote marks ' ' (single quote marks).	`pr -h 'My Novel' file1`
-t	Prevents **pr** from formatting headings and the blank lines at the end of each page.	`pr -t file1`
-s*char*	Separates columns with the character *char* rather than with blank spaces. You must enclose special characters in quote marks.	`pr -s'*' file1`

Figure 2-1 (Part 2 of 2). pr Command Flags

You can use more than one flag with the **pr** command. In the following example, the **pr** command formats the file `efile` with two columns (`-2`), double spacing (`d`), and the title `My Novel` (rather than the name of the file) in the heading:

```
$ pr -2dh 'My Novel' efile
$ _
```

Printing Files—The print Command

Use the **print** command to send one or more files to the system printer. The **print** command places files in a printer queue—a list of files waiting to be printed. Once the **print** command places your files in the queue, you can continue to do other work on your system while you wait for the files to print.

The general form for the **print** command is: **print** *filename*. If your system has more than one printer, you can use a command of the form: **print** *printername filename*, where *printername* is the name of a particular printer.

Note: If your system has more than one printer, one of those printers is the ***default*** printer. When you do not enter *printername*, your print request goes to the default printer.

There are several flags to the **print** command that you may find useful. Some of the flags are explained in Figure 2-2 on page 2-10 and all of them are covered under **print** in *AIX Operating System Commands Reference.*

To Print Files

- At the $ (shell) prompt, enter a command of the form:

  `print` *filename*

  where *filename* is the name of one or more files separated by spaces.

- If your system has more than one printer, you can choose the printer you want with a command of the form:

  `print` *printername filename*

  where *printername* is the name of printer (for example, **lpd**).

The following example first shows how to print a single file, and then two files, with the **print** command:

```
$ print lpd file1
$ print lpd file2 file3
$ _
```

The first **print** command sends the file `file1` to the **lpd** printer, and then returns the $ (shell) prompt to the screen. The second **print** command sends the files `file2` and `file3` to the same print queue, and then returns the shell prompt to the screen before the files finish printing.

Several flags to the **print** command let you control how the files print. The general format for using a flag with the **print** command is:

`print` *flag filename*

Figure 2-2 explains some of the most useful flags to the **print** command. These and other flags are covered under **print** in *AIX Operating System Commands Reference.*

Flag	Action	Example
-ca *filename*	Cancels a print request. (Do not specify *printername.*)	`print -ca file1`
-nc=*num*	Prints *num* copies of the file. Normally, only one copy is printed.	`print -nc=3 file1`
-no	Notifies you when the print job is completed by placing a message on your screen.	`print -no file1`
-q	Displays the status of the print queue and printer.	`print -q`

Figure 2-2 (Part 1 of 2). print Command Flags

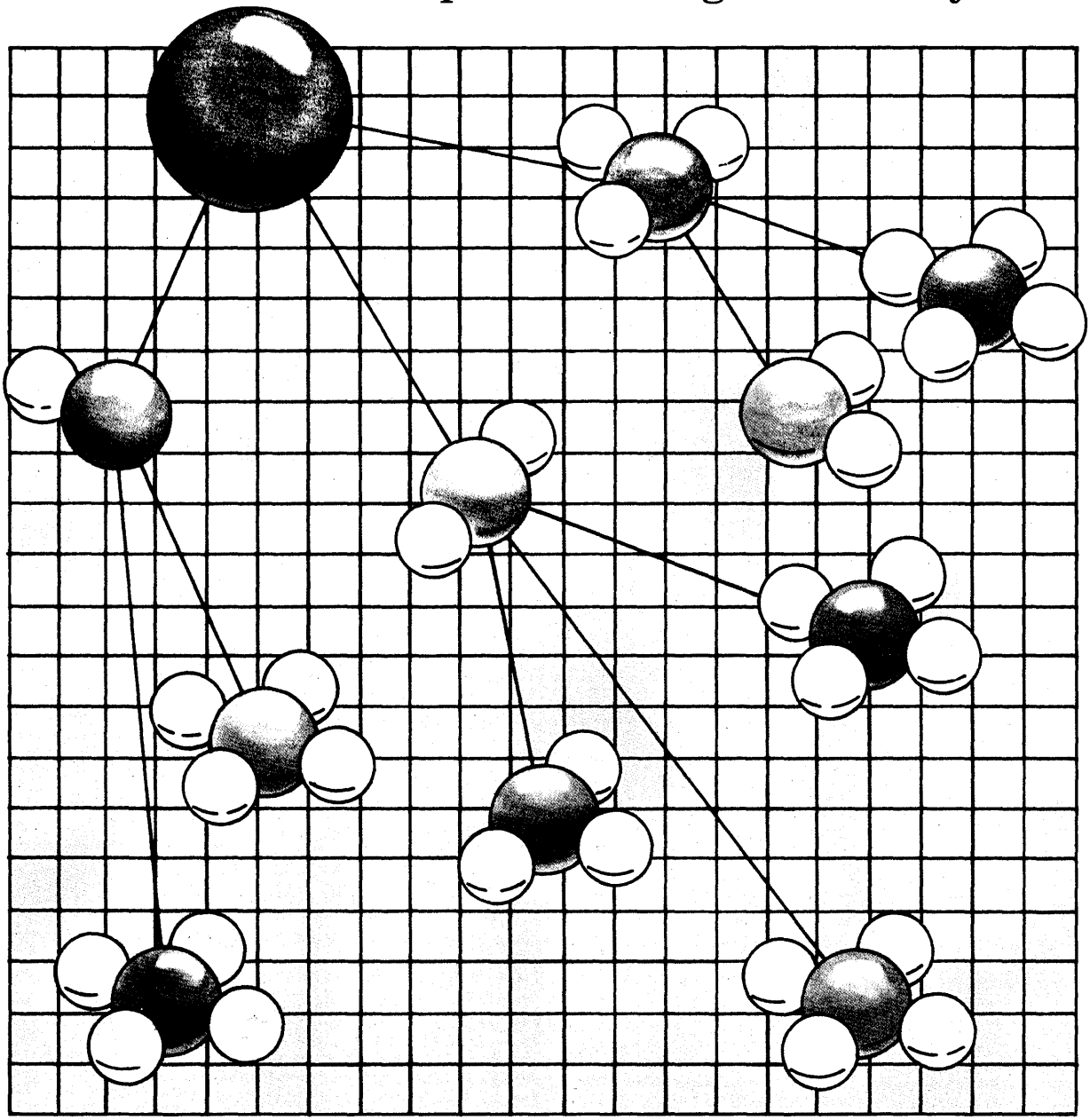
Flag	Action	Example
-tl=*title*	Places *title* on the first page of the printed document and displays *title* when you use the **-q** flag.	`print -tl=Book file1`
-to=*name*	Labels the printout for delivery to this person.	`print -to=fred afile`
-cp	Copies the file. If you want to change a file while you are waiting for the current copy to print, use the **-cp** flag.	`print -cp file1 file2`

Figure 2-2 (Part 2 of 2). print Command Flags

You can specify more than one flag with the **print** command. For more information on printing and printer control, see:

- *Managing the AIX Operating System*
- **piobe** in *AIX Operating System Commands Reference*
- **print** in *AIX Operating System Commands Reference.*

Chapter 3. Using the File System



CONTENTS

About This Chapter

A ***file*** is a collection of data stored together in the computer. A ***file system*** is the arrangement of files into a useful order. This chapter explains the concepts of the AIX file system and the commands you can use to work with it. The material in this chapter can help you design a file system that is appropriate for the type of information you work with and the way you work.

A good way to learn how the file system works is to try the examples in this chapter on your system. When you work through the examples, it is easiest for you to do each example in order. That way, you make the information on your screen consistent with the information in this guide.

This chapter describes file systems that are stored on the AIX fixed-disk. You also can create file systems on diskettes. For information about creating and using diskette file systems, see *Managing the AIX Operating System.*

Note: The examples in this chapter rely on the files created in "Creating Sample Files for This Chapter" on page 2-4.

Understanding Files, Directories, and Path Names

A ***file system*** is an arrangement of many different pieces of information into a useful pattern. Any time you organize information, you create something like a computer file system. For example, the structure of a manual file system (file cabinets, file drawers, and file folders) resembles the structure of a computer file system. Once you have organized your file system, manual or computer, you can find a particular piece of information quickly because you understand the structure of the system.

To understand the AIX file system, you should first become familiar with three concepts:

- **Files and file names**
- **Directories**
- **Tree structure and path names.**

Files and File Names

A file can contain the text of a document, a computer program, records for a general ledger, the numerical or statistical output of a computer program, or other data. A file name can be up to 14 characters long and can contain letters, numbers, and underscores. File names should not include characters that have a special meaning to the shell, including \ (backslash), & (ampersand), and . (period or dot). The characters with special meanings to the shell are described under "Shell Reserved Characters and Words" on page 6-34.

Note: Unlike some operating systems, the AIX Operating System distinguishes between uppercase and lowercase letters in file names, (that is, it is ***case sensitive***). For example, the following three file names specify three different files: `filea`, `FILEA`, and `Filea`.

It is a good idea to use file names that tell you what is in a file. For example, a file name such as `file.memo` gives you a good idea

about the contents of the file. However, a file name such as `a.,,,,.` tells you little or nothing about what the file contains.

It is a good idea to use a consistent pattern to name related files. For example, for a report divided into chapters, one file per chapter, the files might be named:

chap1
chap2
chap3
and so on . . .

Directories

With the RT PC file system, you can organize your files into groups and subgroups (like the cabinets, drawers, and folders in a manual file system). These groups are called ***directories***. A well-organized system of directories (and subdirectories) lets you retrieve and manipulate the data in your files efficiently.

Directories are different from files in two significant ways:

- Directories contain the names of files, other directories, or both.
- Directories are organizational tools (not storage places for data).

You may find it convenient to name files and directories with different conventions.

Note: You can determine whether a file is an ordinary file, a directory, or another type of file with either the **ls -l** command (`ls -l` *filename*) or the **file** command (`file` *filename*).

When you first log in to the system, you are working in your ***login directory***. As you work with the system, you probably will change directories, sometimes frequently. The directory you are working in at any given time is your ***current directory*** (sometimes called the ***working directory***.)

Whenever you want to know the name of your current (working) directory, simply enter the **pwd** (print working directory) command, as the following example shows:

```
$ pwd
/u/uname
$ _
```

Note: Instead of *uname*, you see the name of your login directory.

The **pwd** command returns the ***path name*** of your current directory. Whenever you are uncertain what directory you are working in, or where that directory fits in the system, use the **pwd** command to find out.

File System Structure and Path Names

A file system is an arrangement of directories and files that resembles an upside-down tree (a hierarchy). Figure 3-1 on page 3-7 shows a typical AIX file system. (In this example, directory names are printed in blue and file names are printed in black.)

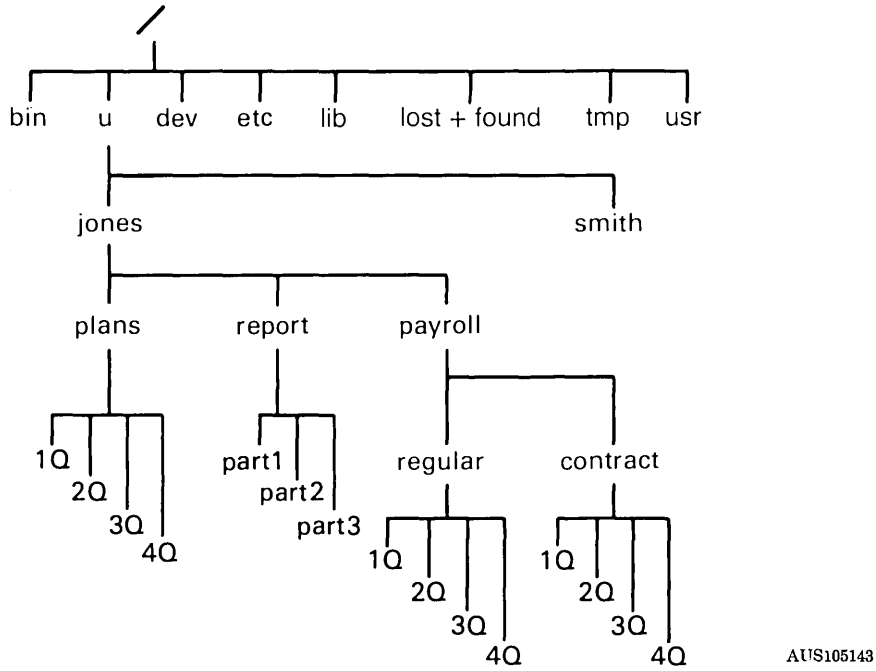


Figure 3-1. A Typical AIX File System

At the top of the file system is a directory called the **root directory**, indicated by the / (slash) symbol. At the next level of the file system are several directories, each of which has its own system of subdirectories and files. At this level, the **u** directory contains the names of the login directories for the users of this system (`smith` and `jones`). At the next level of this file system are the login directories themselves (`smith` and `jones`), the directories where these users begin their work after logging in.

A higher level directory is frequently called a ***parent directory***. For example, in Figure 3-1, the directories `plans` and `payroll` have the same parent directory: `jones`.

A ***path name*** is a sequence of directory names separated by / (slashes) and ending with a file name or a directory name. A path name specifies the location of a directory or file within the file system. The following path name is based on Figure 3-1:

```
/u/jones/report/part3
```

The first / represents the root directory, and indicates the starting place for the search. The remainder of the path name indicates that the search is to go to the u directory, then to directory jones, next to directory report, and finally, to the file part3.

Whether you are changing your current directory, sending data to a file, or copying or moving a file from one place in your file system to another, you use path names to indicate the objects you wish to manipulate.

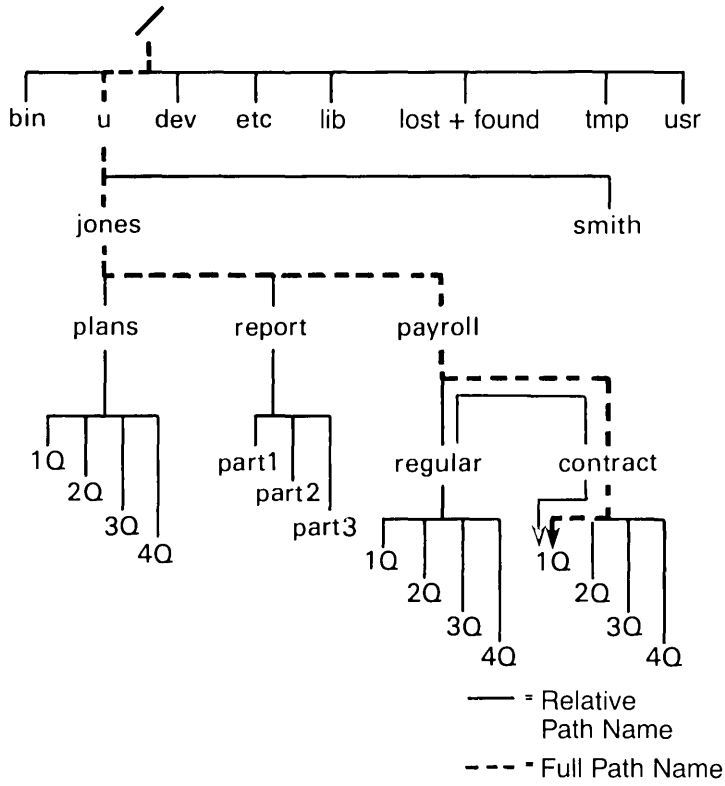
A path name that starts with / (the name for the root directory) is called a ***full path name***. You also can think of a full path name as the full name for a file or directory. Regardless of where you are working in the file system, you can always find a file or directory by specifying its full path name.

Note: If there are other users on your system, you may or may not be able to get to their files and directories, depending upon the permissions set for them. For more information about file and directory permissions, see "Changing Permissions—The chmod (Change Mode) Command" on page 3-45.

The AIX file system also lets you use ***relative path names***. Relative path names do not begin with / (the symbol for the root directory), but are *relative* to the current directory. A relative path name can be either:

- The name of a file in the current directory
- A path name that begins with the name of a directory one level below your current directory
- A path name that begins with .. (**dot dot**, the relative path name for the parent directory).

  Note: Every directory contains at least two entries: .. (**dot dot**) and . (**dot**, which refers to the current directory).



AUS105144

Figure 3-2. Relative and Full Path Names

In Figure 3-2, for example, if your current directory is `jones`, the relative path name for the file `1Q` in directory `contract` is `payroll/contract/1Q`. (Compare this relative path name with the full path name for the same file, `/u/jones/payroll/contract/1Q`, and you will see that relative path names can be quite convenient.)

The rest of this chapter explains how to create and modify the file system and how to use file system commands.

Creating a Directory—The mkdir (Make Directory) Command

Note: Before you can follow the examples in this chapter, you must be logged in to the system and have three files in your login directory. The examples in this chapter use the three files created in "Creating Sample Files for This Chapter" on page 2-4, `file1`, `file2`, and `file3`. (To find out what files you have in your current directory, enter `ls`.) If you use files with different names, make the appropriate substitutions as you work through the examples.

Directories allow you to organize your files into useful groups. For example, you could put all the sections of a report in a directory named `report`, or the data and programs you use in cost estimating in a directory named `estimate`. A directory can contain the names of files, other directories, or both.

To create a directory, use the **mkdir** (make directory) command. The form for the **mkdir** command is `mkdir` *dirname*. Your new directory is created at the next level below your current directory.

To Create a Directory

Enter a command of the form:

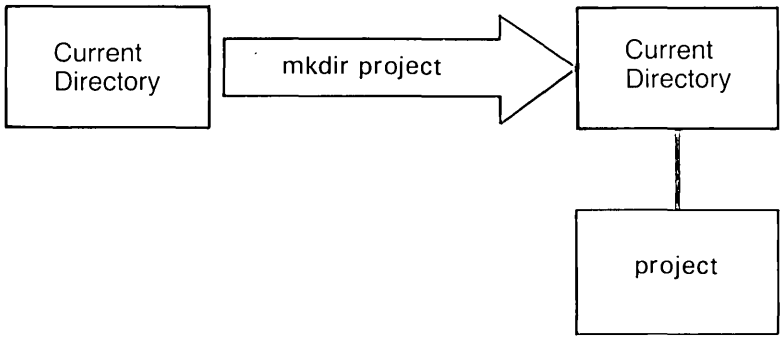
`mkdir` *dirname*

where *dirname* is the name you want to assign to the directory.

To create a directory, type the **mkdir** command followed by the name you want to assign to the directory. The following example shows how to create a directory named `project`:

```
$ mkdir project
$ _
```

The new directory, `project`, is one level below the current directory, as Figure 3-3 on page 3-11 shows.



AUS105145

Figure 3-3. Relationship Between a New Directory and the Current Directory

Like file names, directory names can be up to 14 characters long, and contain letters, numbers, periods, commas, and underscores. Because there is nothing to automatically distinguish a file name from a directory name, you may find it useful to establish naming conventions.

Listing Directory Contents— The ls (List) Command

Use the **ls** (list) command to list the contents of one or more directories. The general form for the **ls** command is simply `ls`, which lists the contents of your current directory.

You can do several other things with the **ls** command as well. For example, you can list the contents of directories other than the current directory. You also can use the **ls** command flags to get different types of information about the contents of a directory as is explained under "ls Command Flags" on page 3-14.

To List Directory Contents

Enter: `ls` *dirname*

where *dirname* is the name of the directory.

The Current Directory

To list the contents of your current directory, enter `ls`:

```
$ ls
file1
file2
file3
project
$ _
```

Used without flags, the **ls** command simply lists the names of the files and directories in your current directory.

Other Directories

To list the contents of directories other than your current directory, use a command of the form `ls` *pathname*. In the following example, the current directory is your login directory, and the **ls** command lists the contents of a different directory, **/u**:

```
$ ls /u
alan
bill
brian
cath
don
fox
george
heinz
jerome
laurie
mark
melh
nanda
pat
stan
thom
zern
$ _
```

As this example shows, the **ls** command ordinarily sorts directory and file names alphabetically.

Note: Your listing of the **/u** directory will not be exactly like this one. It will contain the names of login directories for users on your system. It also may have a different set of system files, depending upon how your system is customized.

ls Command Flags

In its simple form, the **ls** command lists only the names of files and directories. However, there are several flags (or options) that you can use with the **ls** command. Flags to the **ls** command typically give you more information about the listed files and directories or change the way the listing displays. To use flags, use a command of the form `ls` *-flag name(s)*. If you want to use more than one flag, type all of the flag names together in one string as shown in the following example:

```
ls -ltr
```

The following table lists some of the most useful **ls** command flags:

Flag	Action
-l	Lists in **long** format. An **-l** listing provides the type, permissions, number of links, owner, group, size and time of last modification for each file or directory listed.
-t	Sorts the files and directories by the **time** they were last modified (latest first), rather than alphabetically by name.
-a	Lists **a**ll entries. Without this flag, the **ls** command does not list the names of entries that begin with . (period), such as relative directory names, **.profile**, and **.login**.
-r	Reverses the order of the sort to get reverse alphabetic order (**ls -r**) or reverse time order (**ls -tr**).

Figure 3-4. ls Command Options

The following example shows a long (**-l**) listing of a current directory (substitute your user name for *uname*):

```
$ ls -l
total 4
-rw-r--r--   1 uname    system     101 Jun  5 10:03 file1
-rw-r--r--   1 uname    system     171 Jun  5 10:03 file2
-rw-r--r--   1 uname    system     130 Jun  5 10:06 file3
drwxr-xr-x   2 uname    system      32 Jun  5 10:07 project
$ _
```

The **ls -l** command returns the following information:

Field	Information
total 4	Number of 512-byte blocks taken up by files in this directory.
drwxr-xr-x	File type and permissions set for each file or directory. The first character in this field indicates file type: - (hyphen) for ordinary files d for directories b for block special files c for character special files p for pipe (first in, first out) special files. Remaining characters indicate what read, write, and execute permissions are set for owner, group, and others. For more information on permissions, see "Changing Permissions—The chmod (Change Mode) Command" on page 3-45.
1	Number of links to each file. For an explanation of ***file links***, see "Linking Files—The ln (Link) Command" on page 3-27.
uname	User name of the file's owner.
system	Group to which the file belongs.
101	Number of characters in the file.
Jun 5 10:03	Date and time the file was created or last modified.
file1	Name of the file or directory.

Figure 3-5. ls -l Command Information

There are other flags to the **ls** command that you may find useful as you gain experience with the AIX system. All of the **ls** command flags are explained under **ls** in *AIX Operating System Commands Reference.*

Changing Directories—The cd (Change Directory) Command

The **cd** (change directory) command changes your current or working directory. You can access any directory in the file system from any other directory in the file system.

Note: You must have permission to access a directory before you can use the **cd** command to make that directory your current directory. (For more information on directory permissions, see "Protecting Files and Directories" on page 3-42.)

The general form of the **cd** command is `cd` *pathname*. You can use either a full path name or a relative path name with the **cd** command, depending upon the location of the directory.

The **cd** command alone (without a path name) returns you to your login directory.

Whenever you want to check the name of your current directory, enter the **pwd** (print working directory) command.

To Change Your Current Directory

Enter:

`cd` *pathname*

where *pathname* is either the full or relative path name of the directory that you want to make your current directory.

To Return to Your Login Directory

Enter:

`cd`

Note: In the following examples, the term *uname* stands for the name of your login directory.

Displaying the Path Name of Your Current Directory

In the next example, the **pwd** command displays the path name of the current directory (/u/*uname*):

```
$ pwd
/u/uname
$ cd project
$ pwd
/u/uname/project
$ _
```

Next, the **cd** command, with a relative path name, makes project the current directory. Finally, another **pwd** command displays the name of the current directory (/u/*uname*/project), showing that the change has been made.

Returning to Your Login Directory

To return to your login directory from any other directory in the file system, use the **cd** command without a path name:

```
$ cd
$ pwd
/u/uname
$ _
```

To change your current directory to a directory that is either above your current directory or on a different branch of the file system, use an absolute path name with the **cd** command:

```
$ pwd
/u/uname
$ cd /u
$ pwd
/u
$ _
```

Using Relative Directory Names (. and .. Notation)

Every directory contains at least two entries: . ("dot") and .. ("dot dot"), which refer to directories relative to the current directory.

. ("dot") This entry refers to the current directory.

.. ("dot dot") This entry refers to the ***parent directory*** (the directory immediately above the current directory in the file system).

To display the relative directory names, use the **-a** flag with the **ls** command. (For more information about the **ls** command and its flags, see "Listing Directory Contents— The ls (List) Command" on page 3-12.)

In the following example, the **ls** command does not return any information about the contents of the directory `project` because you have created no files or subdirectories in `project`:

```
$ cd uname/project
$ ls
$ ls -a
.
..
$ _
```

However, the **ls -a** command lists the directory entries that begin with . (dot)—the relative directory names.

You can use the relative directory name .. to refer to files and directories that are located above the current directory in the file system. That is, if you want to move up the directory tree one level, you can use the relative directory name for the parent directory rather than using an absolute path name. In the following example, the `cd ..` command changes the current directory from `project` to *uname* (the parent directory of `project`):

```
$ pwd
/u/uname/project
$ cd ..
$ pwd
/u/uname
$ _
```

To move up the directory structure more than one level, you can use a series of relative directory names, for example:

```
$ cd ../..
$ pwd
/
$ _
```

Removing Files—The rm (Remove File) Command

When you no longer need a file, you can remove it with the **rm** (remove file) command. The general format for the **rm** command is rm *filename*. The *filename* can be the name of the file alone, a relative path name, or an absolute path name, depending upon where the file is located in relation to the current directory.

To Remove a File

Enter:

rm *filename*

where *filename* is a simple file name, a path name, or a list of file names.

Multiple file names can be ***linked*** to a single file. The **rm** command removes the links between file names and files. **rm** removes the file itself only when it removes the last link to that file. For more information on links, see "Linking Files—The ln (Link) Command" on page 3-27.

Notes:

1. You also can remove files with the **del** (delete) command. For an explanation of **del**, see **del** in *AIX Operating System Commands Reference.*

2. You must have permission to access a directory before you can remove files from it. (For more information on directory permissions, see "Protecting Files and Directories" on page 3-42.)

Removing Multiple Files

You can remove more than one file at a time with the **rm** command by using the following ***pattern-matching*** characters (characters that can stand for another character or string of characters):

*	Matches any string of characters in a file name.
?	Matches any character in a file name.
[. . .]	Matches any of the enclosed characters.

For example, the pattern-matching string `*.jun` matches any of the following file names: `receivable.jun`, `payable.jun`, `payroll.jun`, and `expenses.jun`. You could remove all four of these files from your current directory with the `rm *.jun` command.

Warning: Be certain that you understand how the * pattern-matching character works before you use it. For example, the **rm *** command removes every file in your current directory. You may choose to use the * character with the **del** command, which lets you verify the files to be removed before removing them.

Similarly, you can use the pattern-matching character ? with the **rm** command to remove files whose names are the same except for a single character. For example, if your current directory contains the files `record1`, `record2`, `record3`, and `record4`, you could remove all four of the files with the `rm record?` command. For more information on pattern-matching characters, see "Matching Patterns" on page 5-13.)

Removing Multiple Files and Directories

Ordinarily, the **rm** command removes only files, not directories. (For information about removing directories, see "Removing Directories—The rmdir (Remove Directory) Command" on page 3-23.) However, you can remove directories with the **-r** (recursive) flag to the **rm** command. When used with the **rm** command, the **-r** flag first deletes the files from a directory and

then deletes the directory itself. The format for the **rm** command with the **-r** format is `rm -r` *pathname.*

Warning: Be certain that you understand how the **-r** works before you use it. For example, the `rm -r *` command would delete all files and directories to which you have access. If you have superuser authority, this command could delete all system files

Another flag to the **rm** command, the **-i** (interactive) flag, allows you to remove files selectively. It is often convenient to use the **-i** flag together with the **-r** flag, for example:

```
rm -ri pathname
```

With the **-i** flag, the **rm** command prompts you for a `y` (yes) or `n` (no) response before it removes either a file or the directory.

Removing Directories—The rmdir (Remove Directory) Command

When you no longer need a particular directory, you can remove it with the remove directory (**rmdir**) command. The **rmdir** command removes only empty directories (ones that contain no files or subdirectories) and cannot remove the current directory. (For information about removing files from directories, see "Removing Files—The rm (Remove File) Command" on page 3-20.) The general format of the **rmdir** command is `rmdir` *dirname.*

To Remove a Directory

1. Make sure the directory is empty.
   a. Enter `ls -a` to list the directory's contents.
   b. Use the **rm** command to remove any files.
   c. Use **rm** or **rmdir** to remove any directories.
2. Use the **cd** command to move to a different directory (usually the parent directory).
3. Enter `rmdir` *dirname* (where *dirname* is the name or path name of the directory you want to remove).

Before working through the examples in this chapter, create three subdirectories in the directory `project`, as the following example shows. First, use the command `cd project` to make `project` your current directory. Next, use the **mkdir** command to create the directories `schedule`, `tasks`, and `costs`. Finally, use the **cd** command to return to your login directory.

```
$ cd project
$ mkdir costs schedule tasks
$ cd
$ _
```

Removing a Directory

The **rmdir** command removes only empty directories. If you try to remove a directory that contains any directory or file names, the **rmdir** command gives you an error message, as the following example shows:

```
$ rmdir project
rmdir:    project not empty
$ _
```

To remove the directory `project`, you first must remove the contents of that directory. In the following example, the **cd** command makes `project` your current directory, and then the **ls** command lists the contents of `project`:

```
$ cd project
$ ls
costs
schedule
tasks
$ rmdir schedule
$ ls
costs
tasks
$ _
```

The command `rmdir schedule` removes the directory `schedule` from the current directory, `project`.

Removing Multiple Directories

You can remove more than one directory at a time with the **rmdir** command by using pattern-matching characters—characters that can stand for any other character or string of characters. The most commonly used pattern-matching characters are:

*	Matches any string of characters in a file name
?	Matches any character in a file name.

For example, the characters *s (a pattern-matching string) match the names of both directories, tasks and costs. It does not match the name schedule. The **rmdir *** command would remove all empty directories from your current directory.

Similarly, the ??s?? string matches the names of tasks and costs. Each of these directory names consists of two characters (matched by ??), the character s, and two more characters (again matched by ??). The rmdir ??s?? command would remove both of these directories from your current directory.

You can use the * and ? pattern-matching characters together. In the following example, the *s?s string matches the two directory names tasks and costs, since both names consist of a string of characters (matched by *), an s, another character (matched by the ?), and a final s.

```
$ rmdir *s?s
$ ls
$ _
```

The **ls** command shows that project contains no entries.

(For more information on pattern-matching characters, see "Matching Patterns" on page 5-13.)

Removing Your Current Directory

Before you can remove your current directory, you must move out of it (for example, into the parent directory of your current directory). Then use the **rmdir** command to remove the directory you were working in.

The directory project is empty. To remove project, first move to your login directory (the parent directory of project). Then, as the example shows, use the **rmdir** command:

```
$ cd
$ rmdir project
$ ls
file1
file2
file3
$ _
```

Use the **ls** command to verify that your login directory no longer contains the directory project.

Linking Files—The ln (Link) Command

A ***link*** is a connection between a file name and the file itself. An ordinary file usually has one link—a link to its original file name. However, you can use the **ln** (link) command to connect a single file to more than one file name at the same time.

You can link files across directories, but not across file systems. You can link directories only if they have the same parent directory.

To Link Files

Enter a command of the form:

`ln` *filename1 filename2*

where:

- *filename1* is the name of an existing file.
- *filename2* is the new file name that you want to link to *filename1*.

Using Links

Links are convenient whenever you need to work with the same data in more than one place. For example, assume that you have a file that contains assembly line production statistics. You use the same file both in a monthly report to your management and in a monthly synopsis that you prepare for the line workers. You can link the file to two different file names (which can be in different directories), for example, `mgmt.stat` and `line.stat`. Because you have only one copy of the file, you save storage space and you do not have to remember to update multiple copies of the same file (that is, both `mgmt.stat` and `line.stat` always contain the same data.)

In the following example, the **ln** command links the new file name checkfile to the existing file, file3:

```
$ ln file3 checkfile
$ _
```

To verify that file3 and checkfile are two names for the same file, use the **pg** (page) command to display first one and then the other:

```
$ pg file3
When you finish entering your text, enter a period
on a line by itself.  Then enter w to write (or save)
a copy of the new file.
$ pg checkfile
When you finish entering your text, enter a period
on a line by itself.  Then enter w to write (or save)
a copy of the new file.
$ _
```

Both file3 and checkfile are names for the same file. Any change that you make to the file (for example, by editing file3) shows up when you access the file by the other name (for example, checkfile).

How Links Work—Understanding File Names and i-numbers

Each file has a unique identification number, called an ***i-number***. The i-number refers to the file itself—data stored at a particular location— rather than to the file name. A directory entry is simply a link between an i-number and a file name. It is this relationship between files and file names that makes it possible for you to link multiple file names to the same physical file (the same i-number).

To display the i-numbers of files in your directory, use the **ls** command with the **-i** (print i-number) flag:

```
$ ls -i
1079 checkfile
1076 file1
1078 file2
1079 file3
$ _
```

The number beside each file name is the i-number for a file. The i-numbers in your listing probably will be different from those shown in this example. However, the important thing to note is that the i-number for `file3` is the same as the one for `checkfile` (the two files linked in the previous example). In this case, the i-number is 1079.

Removing Links

The **rm** (remove file) command (described under "Removing Files—The rm (Remove File) Command" on page 3-20) does not always remove a file. For example, if a file (an i-number) is linked to more than one file name, the **rm** command removes the link between the i-number and that file name, but leaves the file intact. The **rm** command removes a file only when it removes the last link between a name and that file, as Figure 3-6 shows:

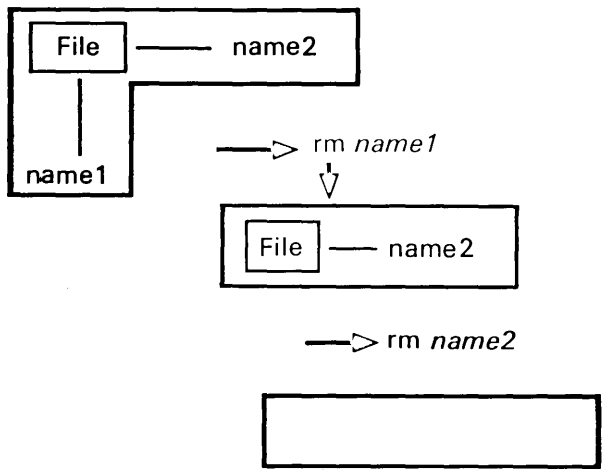


Figure 3-6. Removing Links and Removing Files

To determine how many links there are to a particular i-number, use the **ls** command with the **-i** (print i-number) and **-l** (long listing) flags, as the following example shows:

```
$ ls -il
total 4
 1079 -rw-r--r--   2 uname    system     130 Jun  5 10:06 checkfi
 1076 -rw-r--r--   1 uname    system     101 Jun  5 10:03 file1
 1078 -rw-r--r--   1 uname    system     171 Jun  5 10:03 file2
 1079 -rw-r--r--   2 uname    system     130 Jun  5 10:06 file3
$ _
```

Again, the first number in each entry shows the i-number for that file name. The third field for each entry (the number to the left of the user's name) is the number of links to that i-number. Notice that `file3` and `checkfile` both have the same i-number (1079) and both show two links.

Each time the **rm** command removes a file name, the number of links to that file (i-number) is reduced by one. In the following example, the **rm** command removes the file name `checkfile` and reduces the number of links to i-number 1079 (the same i-number `file3` is linked to) by one:

```
$ rm checkfile
$ ls -il
total 3
 1076 -rw-r--r--   1 uname    system     101 Jun  5 10:03 file1
 1078 -rw-r--r--   1 uname    system     171 Jun  5 10:03 file2
 1079 -rw-r--r--   1 uname    system     130 Jun  5 10:06 file3
$ _
```

Because only one link to i-number 1079 remains, you could use the **rm** command to remove both the name `file3` and the physical file associated with it.

Copying Files—The cp (Copy) Command

The **cp** (copy) command copies files either within your current directory or into some other directory. One use for the **cp** command is to make back up copies of important files. Because the back up and the original are two distinct files, you can work with the original knowing that, if something happens to it—or if you do not want to save your most recent changes—you can start over with the unchanged version. (Compare the **cp** command, which actually copies files, with the **ln** command, explained under "Linking Files—The ln (Link) Command" on page 3-27, which creates multiple names for the same file.)

To Copy Files

- To copy a file, enter:

  cp *source destination*

  Where *source* is the name of the file to be copied and *destination* is the name of the file *source* is to be copied to. (*source* and *destination* can be file names in the current directory or path names.)

- To copy files to a different directory, enter:

  cp *source destination*

  Where *source* is a series of one or more file names and *destination* is a path name that ends with the name of a directory.

Copying Files in the Current directory

Warning: If the destination file exists, the **cp** command erases the contents of that file before it copies the source file. Be certain that you do not need the contents of the destination file, or that you have a back up copy of the file, before you use it as the destination file for the **cp** command.

If the destination file does not exist, the **cp** command creates it. In the following example, the first **ls** command lists the contents of the current directory:

```
$ ls
file1
file2
file3
$ cp file1 file1x
$ ls
file1
file1x
file2
file3
$ _
```

Next, the **cp** command copies the source file, `file1`, into the destination file, `file1x`. Finally, the second **ls** command shows that the directory now contains the file `file1x`.

Copying Files into Other Directories

If you have followed the examples to this point, you do not have a subdirectory in your login directory. You need a subdirectory to do the following examples, so create one with the **mkdir** command:

```
$ mkdir extra
$ _
```

In the following example, the **cp** command copies the file `file2` into directory `extra`:

```
$ cp file2 extra
$ ls extra
file2
$ _
```

The **ls** command lists the contents of `extra`, showing that `extra` directory now contains a copy of the file `file2`.

With the **cp** command, you can copy more than one file at a time into another directory. In the following example, the **cp** command copies `file1` and `file3` into the directory `extra`:

```
$ cp file1 file3 extra
$ ls extra
file1
file2
file3
$ _
```

The **ls** command listing shows that the copies have been made.

To change the name of a file when you copy it into another directory, give the new file name along with the directory name:

```
$ cp file3 extra/notes
$ ls extra
file1
file2
file3
notes
$ _
```

Renaming or Moving Files and Directories—The mv (Move) Command

Use the **mv** (move) command to move one or more files into a different directory or to rename files or directories. The **mv** command can only rename directories; it cannot move them into other directories. The general format of the **mv** command is mv *oldfilename newfilename.*

The **mv** command links a new name to an existing i-number and breaks the link between the old name and that i-number. It is useful to compare the **mv** command with the **ln** command (explained under "Linking Files—The ln (Link) Command" on page 3-27) and the **cp** command (explained under "Copying Files—The cp (Copy) Command" on page 3-31).

Note: When you use **mv** to move files to other directories, it is extremely important to type the name of the destination directory carefully. If you do not supply a valid destination directory name, **mv** will simply rename the file within the same directory, using the invalid directory name as a file name.

To Rename Files and Directories

Enter a command of the form:

mv *oldname newname*

Where *oldname* and *newname* can be names of files in the current directory or path names.

Renaming Files

In the following example, the first **ls** command lists the i-numbers for each of the files in the current directory. (If you work this example on your system, note the i-number for `file1x`; it probably will be a different number than the one shown in the example.) The **mv** command then changes the name of the file `file1x` to `newfile`:

```
$ ls -i
1085 extra
1076 file1
1088 file1x
1078 file2
1079 file3
$ mv file1x newfile
$ ls -i
1085 extra
1076 file1
1078 file2
1079 file3
1088 newfile
$ _
```

You should notice two things in this example. First, the **mv** command removes the entry `file1x` and adds the entry `newfile`. Second, the i-number for the original file (`file1x`) and `newfile` is the same—1088. The **mv** command removes the link between i-number 1088 and file name `file1x`, and replaces it with a link between i-number 1088 and file name `newfile`. No change is made to the file itself.

Renaming Directories

You can use the **mv** command to rename directories, but only when those directories have the same parent directory. For example, as Figure 3-7 on page 3-36 shows, you could use the **mv** command to change the name of the directory `/u/jones/plans` to `/u/jones/schemes` (since both `plans` and `schemes` have the same parent directory, `/u/jones`).

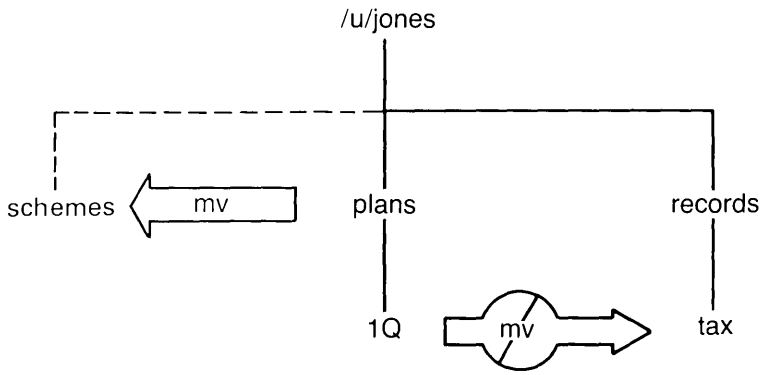


Figure 3-7. Directories That Can and Cannot Be Renamed

However, you could not change the name of the directory `/u/jones/plans/1Q` to `/u/jones/records/tax` with the **mv** command, since `1Q` and `tax` have different parent directories. (To create the directory `/u/tom/memos`, you would use the **mkdir** command, and then use the **mv** command to move the files into it.)

In the following example, the `ls -i` command lists the i-numbers for all entries in the current directory. (Note the i-number for `extra`.) The **mv** command then changes the name of `extra` to `change`:

```
$ ls -i
1085 extra
1076 file1
1078 file2
1079 file3
1088 newfile
$ mv extra change
$ ls -i
1085 change
1076 file1
1078 file2
1079 file3
1088 newfile
$ _
```

Notice that the `ls -i` command does not list the original directory name `extra`. It does list the new directory name, `change`, and it shows that the i-number for the new directory name is the same as it was for the original directory name (1085 in this example).

Moving Files to a Different Directory

The **mv** command also can move one or more files to a different directory.

Note: Type the directory name carefully. The **mv** command does not distinguish between file names and directory names. If you provide an invalid directory name, the **mv** command simply takes that name as a new file name. The result is that the file is renamed rather than being moved.

In the following example, the **ls** command lists the entries in the current directory. Then the **mv** command moves three files from the current directory to the directory `change`.

```
$ ls
change
file1
file2
file3
newfile
$ mv file1 file2 change
$ ls
change
file3
newfile
$ ls change
file1
file2
file3
notes
$ _
```

The second **ls** command shows that the current directory no longer contains the files `file1` or `file2`. Those files now are in the directory `change`, as the `ls change` command shows.

Backing up and Restoring Files

Files and directories represent a significant investment of time and effort. At the same time, all computer files are potentially easy to change or erase, intentionally or by accident. A ***backup copy*** of a file is a duplicate of that file stored on a different storage medium (diskette or tape). When you have a recent backup copy of a file, you have a good place to start over if your original file is damaged or lost. To make backup copies of individual files, use the **backup -i** command.

Note: The **backup -i** command backs up the specified files onto a diskette. A diskette must be ***formatted*** before you can use it as a backup medium. Formatting erases any data stored on the diskette.

To Format a Diskette

1. Enter:

   ```
   format
   ```

   The system displays the message:

   ```
   Insert a new diskette for /dev/fd0
   and strike ENTER when ready
   ```

2. Insert a diskette in diskette drive 0 (A) and press **Enter.**

   The system displays the message:

   ```
   Formatting . . .
   ```

   and then, when the diskette is formatted, the message:

   ```
   Formatting . . . Format completed
   ```

The diskette is ready to be used.

To Back up Individual Files

1. Enter:

   ```
   backup -i
   ```

2. When the system displays the prompt:

   ```
   Please mount volume 1 on /dev/rfd0
   ... and type return to continue _
   ```

   insert a formatted diskette into diskette drive 0 (A) and press **Enter**.

3. When the cursor returns to the left side of the screen, enter one file name per line until you have entered the names of all the files to be backed up. (Be certain to press **Enter** after each file name, including the last file one).

4. Press **END OF FILE**.

5. When the backup is complete, you receive a message of the form:

   Done at *day date time year*
   n blocks on *n* volume(s)

   Remove the diskette, label it clearly, and store it in a safe place.

The **backup** command copies files in a compact form that is not directly usable. To make the backup copies usable, use the **restore** command to transfer them from the diskette to the system.

To Restore Individual Files

1. Insert the backup diskette into diskette drive 0.
2. Enter a command of the form:

   `restore -x` *filename*

Making backup copies of individual files is important and should be a routine part of your work. However, your backup routine also should include regular backups of complete file systems. For information about backing up file systems, see *Managing the AIX Operating System.* Also see **backup** and **restore** in *AIX Operating System Commands Reference.*

Protecting Files and Directories

The two most common reasons for protecting files and directories are:

- They contain sensitive information that should not be available to everyone who uses your system.
- Not everyone who has access to them should have the power to alter them.

To Check and Set Permissions

1. Enter `ls -l` *filename* to display the current permissions.
2. Use the **chmod** command to change permissions, if necessary.
3. Use the **chown** command to change the ***owner***, if necessary.
4. Use the **chgrp** command to change the ***group***, if necessary.

You can protect a file or directory by setting its ***permissions***—codes that determine how the file can be used by anyone who works on your system. **Warning:** Two or more users can be making changes to the same file at the same time (with an editing program, for example) without realizing it. The changes made by the last user to close the file are saved; those of the other users are lost. Therefore, it is good practice to use permissions to allow only authorized users to modify files, and for those users to communicate with each other about how and when the file is being used.

All files (including directories) have nine permissions associated with them:

- Three types of permissions:

  r (read)
  w (write)
  x (execute)

- For each of three classes of users:

  u (user/owner)
  g (group)
  o (all others)

If you are the owner of a file (usually the person who created it), you can change file permissions with the **chmod** command (described under "Changing Permissions—The chmod (Change Mode) Command" on page 3-45).

The meanings of the three permissions differ slightly between ordinary files and directories, as Figure 3-8 shows.

Permission	For a File	For a Directory
r (read)	Contents can be viewed or printed.	Contents can be read, but not searched. Normally **r** and **x** are used together.
w (write)	Contents can be changed or deleted.	Entries can be added or removed.
x (eXecute)	File can be used as a command (program).	Directory can be searched.

Figure 3-8. Differences Between File and Directory Permissions

Displaying File Permissions

To display the permissions for all of the files in your current directory, use the **ls -l** command:

```
$ ls -l
total 3
drwxr-xr-x   2 uname     system       96 Jun  5 11:08 change
-rw-r--r--   1 uname     system      130 Jun  5 10:06 file3
-rw-r--r--   1 uname     system      101 Jun  5 10:44 newfile
$ _
```

The first string of each entry in the directory (for example, `-rw-r--r--`) shows the permissions for that file or directory. Also note that the third field shows the file's **owner** and the fourth field shows the **group** to which the owner belongs.

You also can use the **ls -l** command to list the permissions for a single file or the **ls -ld** command to list the permissions for a single directory:

```
$ ls -l file3
-rw-r--r--   1 uname     system      130 Jun  5 10:06 file3
$ ls -ld change
drwxr-xr-x   2 uname     system       96 Jun  5 11:08 change
$ _
```

Together, permissions for a file or directory are called its ***permission code***. As Figure 3-9 on page 3-45 shows, a permission code consists of four parts:

- A character that shows file type (- for an ordinary file; **d** for a directory; **b** for a block special file; **c** for a character special file; and **p** for a pipe, or first in, first out, special file)
- A three-character ***permission field*** that shows **user** (owner) permissions
- A three-character permission field that shows **group** permissions
- A three-character permission field that shows permissions for all **others**.

-dbcps	rwx	rwx	rwx
file type	owner permissions	group permissions	permissions for all others

Figure 3-9. File and Directory Permission Fields

When you create a file or directory, the system automatically supplies a predetermined permission code. Typical permission codes are:

- For files:

```
-rw-r--r--
```

- For directories:

```
drwxr-xr-x
```

The - (hyphens) in some positions indicate that those permissions are not allowed.

To change the predetermined permission code, you must change your ***file-creation mode mask*** with the **umask** (set file-creation mode mask) command. For an explanation of **umask**, see **umask** in *AIX Operating System Commands Reference.*

Changing Permissions—The chmod (Change Mode) Command

The **chmod** (change mode) command changes the permissions for files or directories. Your ability to change permissions gives you a great deal of control over the way your data can be used.

For example, you can permit yourself to read and modify a file, permit members of your group to read the file, and prohibit all other system users from any access to the file. You must be the owner of the file or directory before you can change its permissions (that is, your user name must be in the third field of an **ls -l** listing of that file).

There are two ways to specify the permissions set by the **chmod** command:

- With letters and operation symbols
- With octal numbers.

To Change File and Directory Permissions

- Enter a command of the form:

  chmod *group-operation-permission filename*

OR

- Enter a command of the form:

  chmod *octalnumber filename*

Specifying Permissions with Letters and Operation Symbols: The general format of the **chmod** command is chmod *group operation permission filename* where:

- *group* is one of the following:

  u for user (owner)
  g for group
  o for all others (besides owner and group)
  a for all (user, group, and all others)

- *operation* is one of the following:

  + (add permission)
  - (remove permission)
  = (assign permission regardless of previous setting).

- *permission* is one or more of the following:

  r for *read*
  s for *set user or group ID*
  t for *save text in virtual memory*
  w for *write*
  x for *execute*

In the following example, the `ls -l` command displays the permissions for the file `newfile`, and then the command `chmod go+w` gives both the group (`g`) and all other system users (`o`) write permission (`w`):

```
$ ls -l newfile
-rw-r--r--   1 uname    system      101 Jun  5 10:44 newfile
$ chmod go+w newfile
$ ls -l newfile
-rw-rw-rw-   1 uname    system      101 Jun  5 10:44 newfile
$ _
```

The second `ls -l` command displays the new permissions for the file `newfile`.

The procedure for changing directory permissions is the same as that for changing file permissions. However, to list the information about a directory, you use the **ls -ld** command:

```
$ ls -ld change
drwxr-xr-x   2 uname    system       96 Jun  5 11:08 change
$ chmod g+w change
$ ls -ld change
drwxrwxr-x   2 uname    system       96 Jun  5 11:08 change
$ _
```

In this example, the command `chmod g+w` gives the group (`g`) write permission (`w`) for the directory `change`.

If you want to make the same change to the permissions of all entries in a directory, you can use the pattern-matching character * (asterisk) with the **chmod** command. (For more information about pattern-matching characters, see "Shell Reserved Characters and Words" on page 6-34.) In the following example, the command `chmod g+x *` gives execute (`x`) permission to all groups (`g`) for all files (*) in the current directory:

```
$ chmod g+x *
$ ls -l
total 3
drwxrwxr-x   2 uname    system      96 Jun  5 11:08 change
-rw-r-xr--   1 uname    system     130 Jun  5 10:06 file3
-rw-rwxrw-   1 uname    system     101 Jun  5 10:44 newfile
$ _
```

The `ls -l` command listing shows that the group now has execute permission (`x`) for all files in the current directory.

An ***absolute*** permission assignment resets all permissions for a file (or files), regardless of how the permissions were set previously. In the following example, the `ls -l` command lists the permissions for the file `file3`, and then the command `chmod a=rwx` gives all three permissions (`rwx`) to all users (`a`):

```
$ ls -l file3
-rw-r-xr--   1 uname    system     130 Jun  5 10:06 file3
$ chmod a=rwx file3
$ ls -l file3
-rwxrwxrwx   1 uname    system     130 Jun  5 10:06 file3
$ _
```

You can also use an absolute assignment (`=`) to remove permissions. In the following example, the command `chmod a=rw- newfile` removes execute permission (`x`) for all groups (`a`) from the file `file3`:

```
$ chmod a=rw- file3
$ ls -l file3
-rw-rw-rw-   1 uname    system     130 Jun  5 10:06 file3
$ _
```

Specifying Permissions with Octal Numbers: Another way to specify the permissions with the **chmod** command is with ***octal numbers***. An octal number corresponds to each type of permission:

4 = **read**

2 = **write**

1 = execute

To specify a group of permissions (a permissions field), add together the appropriate octal numbers, for example:

3 = -wx (2 + 1)

6 = rw- (4 + 2)

7 = rwx (4 + 2 + 1)

0 = --- (no permissions)

The entire permission code for a file or directory is specified with a three-digit octal number, one digit each for **owner**, **group**, and **others**. The following table shows how octal numbers relate to permission fields:

Octal Number	Owner Field	Group Field	Others Field	Complete Code
777	rwx	rwx	rwx	rwxrwxrwx
755	rwx	r-x	r-x	rwxr-xr-x
700	rwx	---	---	rwx------
666	rw-	rw-	rw-	rw-rw-rw-

Figure 3-10. How Octal Numbers Relate to Permission Fields

To use octal number permission codes with the **chmod** command, enter a command of the form: `chmod` *octalnumber filename*, as the following example shows:

```
$ ls -l file3
-rw-rw-rw-   1 uname    system     130 Jun  5 10:06 file3
$ chmod 754 file3
$ ls -l file3
-rwxr-xr--   1 uname    system     130 Jun  5 10:06 file3
$ _
```

It is more difficult to learn to specify permissions with octal numbers than it is to specify them with letters. However, once you

are familiar with the octal number system, you probably will find it more efficient.

Changing Owners and Groups

In addition to setting permissions, you can control how a file or directory is used by changing its owner or group. Use the **chown** command to change the owner and the **chgrp** command to change the group.

To Change the Owner of a File or Directory

Enter:

chown *owner file*

where:

- *owner* is the user name of the new owner.
- *file* is a list of one or more files whose owner you want to change.

To Change the Group of a File or Directory

Enter:

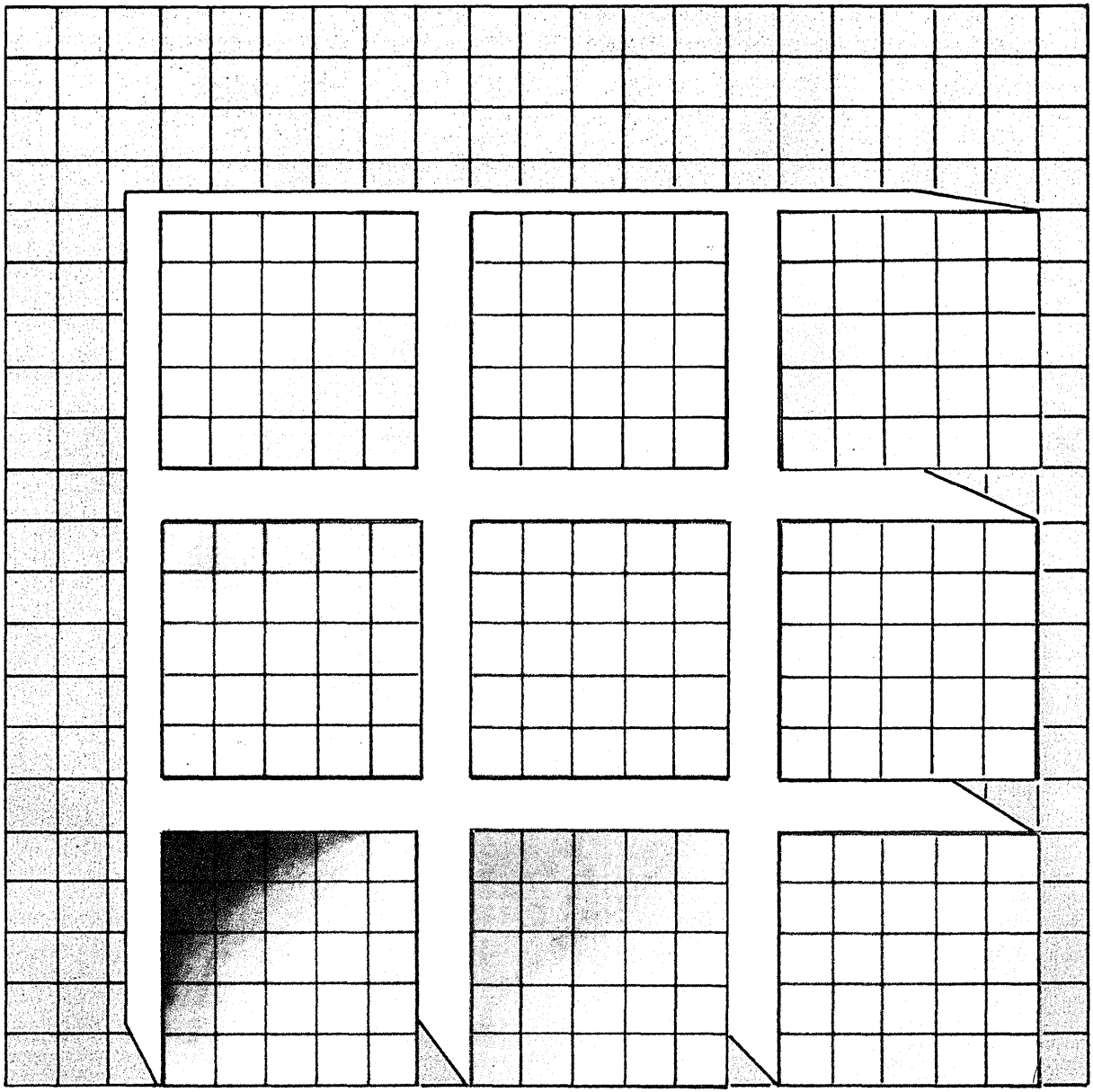
chgrp *group file*

where:

- *group* is the the group ID or group name of the new group.
- *file* is a list of one or more files whose owner you want to change.

For more information about the **chown** and **chgrp** commands, see **chown** and **chgrp** in *AIX Operating System Commands Reference.*

Chapter 4. Understanding Processes



CONTENTS

About This Chapter

This chapter explains the concept of a process and the ways you can run a process, check the status of a running process, and stop a process. Once you understand the material in this chapter, you should be able to use the techniques described in Chapter 5, "Using the Shell with Processes" on page 5-1 to create and control more complex processes.

A good way to learn how processes work is to try the examples in this chapter on your system. In the examples, everything you should type is shaded in blue (for example, **ps**). When you are told in the text to enter a command name or a string of characters, type the characters and then press **Enter**.

Understanding Programs and Processes

A ***program*** is a set of instructions that a computer can interpret and run. You may think of most programs as belonging to one of two categories: application programs (for example, text editors, accounting packages, or electronic spreadsheets) and programs that are components of the AIX Operating System (for example, commands, the shell, and your login procedure). While a program is running, it is called a ***process***.

The AIX Operating System can run a number of different processes at the same time. When more than one process is running, a scheduler built into the operating system gives each process its fair share of the computer, according to established priorities. (Only the person who manages your system can raise these priorities, but any user can lower priorities by using the **nice** command explained under **nice** in *AIX Operating System Commands Reference.*)

The rest of this chapter explains how to:

- Check the status of processes
- Cancel processes
- Run processes in the background.
  - Check the status of background processes
  - Cancel background processes.

This chapter is an introduction to processes. For more detailed information about processes and how to control them, see Chapter 5, "Using the Shell with Processes" on page 5-1.

Checking Process Status—The ps (Process Status) Command

While a program runs, it is called a ***process***. For example, a file on the system contains the program for the **cp** command (a command that copies files). When you enter the **cp** command, you start a process. That **cp** process runs until the system displays the path name of your current directory. Once the path name is displayed, the process ends.

Any time the system is running, several processes are running as well. You can use the **ps** (process status) command to find out what processes are running and to get information about those processes.

To Check Process Status

Enter: `ps`

In the following example, the **ps** command displays the status of all processes associated with your display station:

```
$ ps
   PID   TTY    TIME COMMAND
    39 console  0:10 sh
    43 console  0:03 ps
    32 console  0:01 qdaemon
$ _
```

The **ps** command displays the following information about each process:

PID Process identification. The system assigns a ***process identification number*** (PID number) to each process when that process starts. There is no relationship between a process and a particular PID number; that is, if you start the same process several times, it will have a different PID number each time.

TTY Terminal designation. On a system with more than one terminal, this field tells you which terminal started the process. On a system with only one display station, this field can contain the designation **console** or the designation for one or more virtual terminals.

TIME Time devoted to this process by the computer (given in minutes and seconds as of when you enter **ps**).

COMMAND The name of the command (or program) that started the process. (In this example, `sh` is the **shell** program, **ps** is the process status command that displayed this information, and `qdaemon` is a program that lets you send data to the printer.)

Generally, the simple **ps** command described here tells you all you need to know about processes. However, you can control the type of information that the **ps** command returns by using its flags. One of the most useful **ps** flags is **-e**, which causes **ps** to return information about all processes (not just those associated with your display station). For an explanation of the **ps** command flags, see **ps** in *AIX Operating System Commands Reference.*

Canceling a Process

If you start a process and then decide you do not want to let it finish, you can cancel it by pressing **INTERRUPT**.

Note: **INTERRUPT** does not cancel ***background*** processes. To cancel a background process, you must use the procedure described under "Ending a Background Process—The kill Command" on page 4-13.

To Cancel a Running Process

Press:

INTERRUPT

Note: Most simple AIX commands are not good examples for demonstrating how to cancel a process—they run so quickly that they finish before you have time to cancel them. Therefore, the examples in the rest of this chapter use a command that takes more than a few seconds to run: `find / -type f -print`. This command displays the path names for all files on your system. You do not need to study the **find** in order to complete this chapter—it is used here simply to demonstrate how to work with processes. However, if you want to learn more about the **find** command, see **find** in *AIX Operating System Commands Reference.*

In the following example, the **find** command starts a process. After the process runs for a few seconds, you can cancel it by pressing **INTERRUPT**:

```
$ find / -type f -print
/usr/lib/acct/acctcms
/usr/lib/acct/acctcon1
/usr/lib/acct/acctcon2
/usr/lib/acct/acctdisk
/usr/lib/acct/acctmerg
/usr/lib/acct/accton
/usr/lib/acct/acctprc1
/usr/lib/acct/acctprc2
/usr/lib/acct/acctwtmp
/usr/lib/acct/chargefee
/usr/lib/acct/ckpacct
/usr/lib/acct/dodisk
/usr/lib/acct/fwtmp
/usr/lib/acct/lastlogin
/usr/lib/acct/monacct
/usr/lib/acct/nulladm
/usr/lib/acct/prctmp
/usr/lib/acct/prdaily
/usr/lib/acct/prtacct
/usr/lib/acct/runacct
/usr/lib/acct/sdisk
/usr/lib/acct/shutacct                    <INTERRUPT>
$ _
```

The system returns the $ (shell) prompt to the screen. Now you can enter another command.

Redirecting Input and Output

A command usually reads its input from the keyboard (***standard input***) and writes its output to the display (***standard output***). Often, though, you may want a command to read its input from a file, write its output to a file, or both. With the following shell notation, you can select input and output files for a command:

Notation	Action	Example
<	Reads standard input from a file.	`wc <file3`
>	Writes standard output to a file.	`ls >file3`
> >	Adds standard output to the end of a file.	`ls >>file3`

Figure 4-1. Shell Notation for Reading Input and Redirecting Output

This section explains how to read input from a file and how to write output to a file.

Reading Input from a File—The < Symbol

Use the < (less than) symbol to take input from a file, as the following example shows:

```
$ wc <file3
       3       27      129
$ _
```

The **wc** (word count) command counts the number of lines, words, and characters in the named file. If you do not supply an argument, the **wc** command reads its input from the keyboard. However, in this example, input for **wc** comes from the file named `file3`.

Note: The **wc** command has three flags, **-l** (line count only), **-w** (word count only), and **-c** (character count only), which you can use separately or in combination with each other.

Redirecting Output—The > and > > Symbols

To send output to a file, use either the > (greater than) or >> symbol. The > symbol causes the shell to replace the contents of the file with the output of the command; the shell deletes the contents of the original file. The >> symbol adds (appends) the output of the command to the end of a file. If you use > or >> to write output to a file that does not exist, the shell creates the file.

In the next example, the output of **ls** goes to the file named `file`:

```
$ ls >file
$ _
```

If the file already exists, the shell replaces its contents with the output of **ls**. If `file` does not exist, the shell creates it.

In the following example, the shell adds the output of **ls** to the end of the file named `file`:

```
$ ls >>file
$ _
```

If `file` does not exist, the shell creates it.

In addition to their standard output, processes often produce error or status messages known as ***diagnostic output***. For information about redirecting diagnostic output, see "Standard Error and Other Output" on page 6-29.

Running Background Processes—The & Operator

Besides allowing the processes of several users to run at the same time, the AIX Operating System allows a single user to run more than one process at a time. The & (ampersand) operator at the end of a command tells the system to run that command in the background. Once a process is running in the background, you can enter other commands at your display station.

To Run a Background Process

Enter:

command name&.

Starting a Background Process

Generally, background processes are most useful with commands that take a long time to run. However, because they increase the total amount of work the processor is doing, background processes slow down the rest of the system. This may or may not be a problem, depending upon how much the system slows and the nature of the other work you do while background processes run.

Note: You can use the **nice** command to lower the priority of a process, even a background process. For information about the **nice** command, see **nice** in *AIX Operating System Commands Reference.*

Most processes direct their output to standard output, even when they run in the background. Unless redirected, standard output goes to the display station. Because the output from a background process can interfere with your other work on the system, it is usually good practice to redirect the output of a background process to a file or a printer. Then you can look at the output whenever you are ready.

In the following example, the **find** command runs in the background (&) and directs its output to a file named `dir.paths` (with the > operator):

```
$ find / -type f -print  >dir.paths &
24
$ _
```

When the background process starts, the system assigns it a PID number (24 in this example), displays the number, and then prompts you for another command.

Note: Your process numbers probably will be different from the ones shown in these examples.

(For more information about redirecting output, see "Redirecting Input and Output" on page 4-9.)

Checking Background Process Status

As long as a background process is running, you can check its status with the process status (**ps**) command explained under "Checking Process Status—The ps (Process Status) Command" on page 4-5. You can also check the status of a particular process by using the **-p** flag and the PID number with the **ps** command (for example, `ps -p` *PID number*).

The following example shows how to start another **find** process and then check its status:

```
$ find / -type f -print >dir.paths &
25
$ ps -p 25
   PID   TTY    TIME COMMAND
    25 console  0:40 find
$ _
```

For an explanation of the data that the **ps** command displays, see "Checking Process Status—The ps (Process Status) Command" on page 4-5.

You can check background process status as often as you like while the process runs. In the following example, the **ps** command displays the status of the **find** process five times:

```
$ find / -type f -print >dir.paths &
28
$ ps -p 28
   PID   TTY    TIME COMMAND
    28 console  0:18 find
$ ps -p 28
   PID   TTY    TIME COMMAND
    28 console  0:29 find
$ ps -p 28
   PID   TTY    TIME COMMAND
    28 console  0:49 find
$ ps -p 28
   PID   TTY    TIME COMMAND
    28 console  0:58 find
$ ps -p 28
   PID   TTY    TIME COMMAND
    28 console  1:02 find
$ ps -p 28
   PID   TTY    TIME COMMAND
$ _
```

Notice that the sixth **ps** command returns no status information (because the **find** process ended before the last **ps** command was entered).

Ending a Background Process—The kill Command

If you decide, after starting a background process, that you do not want the process to finish, you can cancel the process with the **kill** command. Before you can cancel a background process, you must know its PID number. (If you have forgotten the PID number of that process, use the **ps** command, described under "Checking Process Status—The ps (Process Status) Command" on page 4-5, to list the PID numbers of all processes.)

Note: If you want to end all of the processes you have started, use the **kill 0** command. You do not have to know the PID numbers to use **kill 0**.

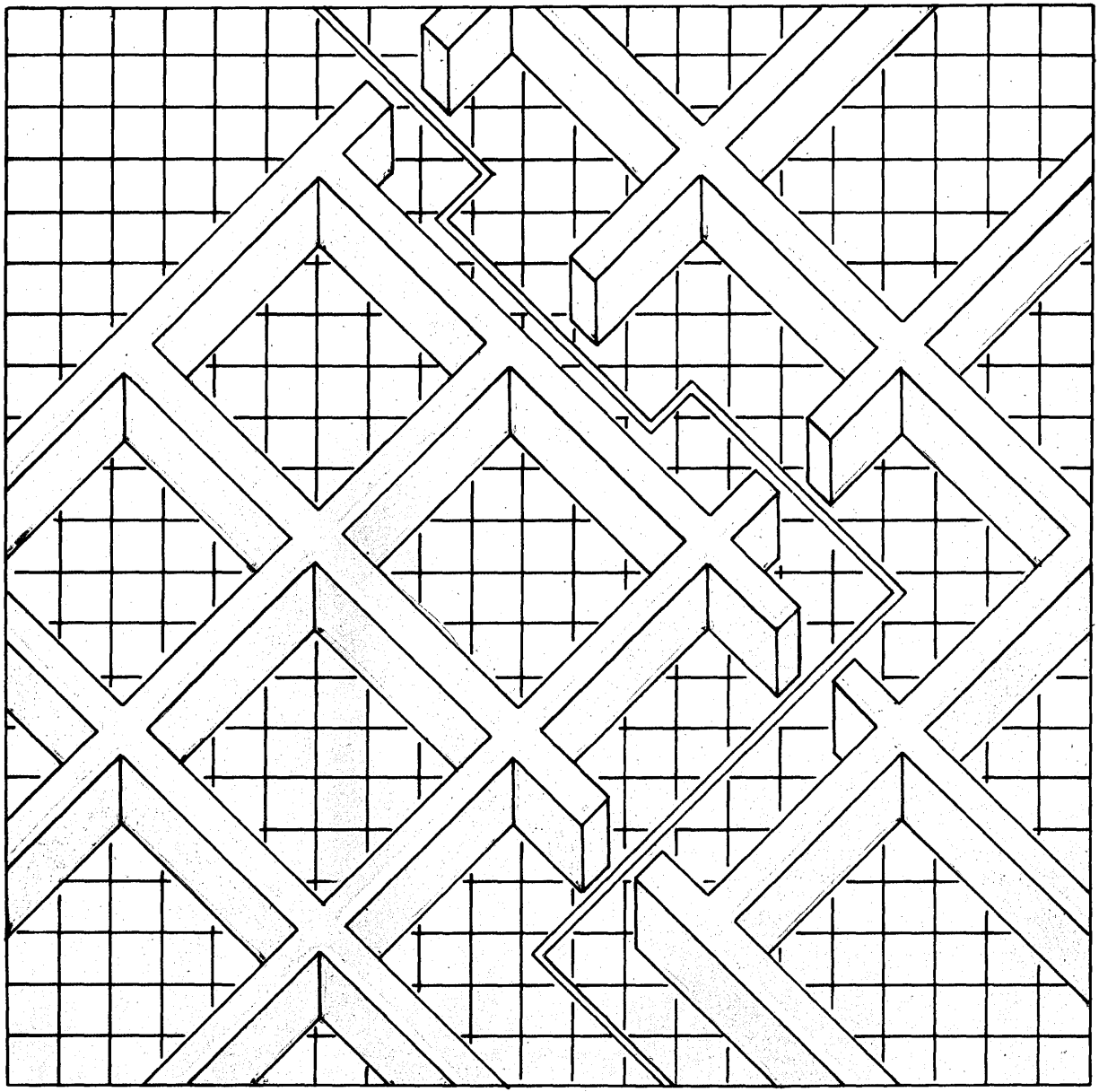
In the following example, after the **find** starts, the **ps** command (without the **-p** flag) displays the status of all processes:

```
$ find / -type f -print >dir.paths &
38
$ ps
   PID   TTY    TIME COMMAND
    20 console  0:11 sh
    38 console  0:10 find
    16 console  0:01 qdaemon
    39 console  0:03 ps
$ kill 38                          [or kill 0]
$ ps
38 Terminated
   PID   TTY    TIME COMMAND
    20 console  0:11 sh
    16 console  0:01 qdaemon
    41 console  0:03 ps
$ _
```

The command **kill 38** stops the background **find** process, and the second **ps** command returns no status information about PID number 38. The system does not display the termination message until you enter your next command (unless that command is **cd**).

Note: In this example, `kill 38` and `kill 0` have the same effect because only one process has been started from this display station.

Chapter 5. Using the Shell with Processes



CONTENTS

About This Chapter

The shell is a program that interprets commands for the AIX Operating System and provides a command programming language. All of the commands described so far in this book are interpreted by the shell; however, you do not have to know anything about the shell in order to use them. In contrast, this chapter explains some tasks you can do by using the shell explicitly:

- Connect commands to each other
- Use multiple commands and groups of commands
- Use special characters to match file names
- Write shell procedures (programs).

Chapter 6, "Using Advanced Shell Features—A Reference" on page 6-1 contains additional information about the shell.

Using Pipes and Filters

A ***pipe*** is a one-way connection between two related commands. One command writes its output to the pipe and the other process reads its input from the pipe. When two or more commands are connected by the | (pipe) operator, they form a ***pipeline.*** Each command in a pipeline runs as a separate process. Figure 5-1 represents the flow of input and output through a pipeline: the output of the first command (Command 1) is the input for the second command (Cmd 2); the output of the second command is the input for the third command (Cmd 3).

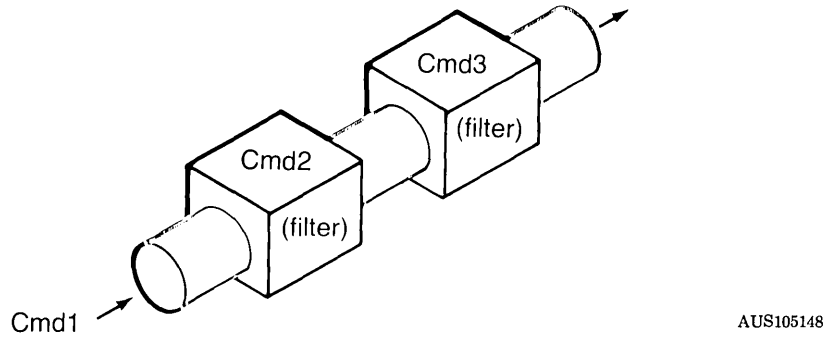


Figure 5-1. Flow Through a Pipeline

A ***filter*** is a command that reads its standard input, transforms that input, and then writes the transformed input to standard output. Filters are typically used as intermediate commands in pipelines, that is, they are connected by a | (pipe) operator, for example, `ls -R|pg` (the **-R** flag causes **ls** to list ***recursively*** the contents of all directories from the current, or named, directory to the bottom of the hierarchy).

Certain commands that are not filters have a flag that causes them to act like filters. For example, the **diff** (compare files) command ordinarily compares two files and writes their differences to standard output. The usual format for **diff** is:

`diff` *file1 file2*

However, if you use the - (hyphen) flag in place of one of the file names, **diff** reads standard input and compares it to the named file. In the following pipeline, **ls** writes the contents of the current directory to standard output; **diff** compares the output of **ls** with the contents of a file named `dirfile`, and writes the differences to standard output one page at a time (with the **pg** command):

```
ls | diff - dirfile | pg
```

In the following pipeline, the standard output of `ls -l /` becomes the standard input to `grep r-x`, a command (in this case a filter) that searches its standard input for a string (r-x) and writes all lines that contain the pattern to its standard output. The standard output of `grep r-x` becomes the standard input to `wc` (which counts the number of lines, words, and characters in its standard input):

```
$ ls -l / | grep r-x | wc
     12     108     717
$ _
```

To get the same results without using a pipeline, you would first have to direct the output of `ls -l/` to a file (for example, `ls -l / >file`). Next, you would have to use that file as input for `grep r-x` and redirect the output of `grep` to another file (for example, `grep r-x <file >file.0`). Finally, you would have to use the output file of `grep` as input for `wc` (for example, `wc <file.0`). The pipeline is much more efficient.

Pipelines operate in one direction only (left to right), and all processes in a pipeline can run at the same time. A process pauses when it has no input to read or when the pipe to the next process is full.

Using Multiple Commands and Command Lists

The shell usually takes the first word on a command line as the name of a command, and then takes any other words as arguments to that command. However, the following operators give you five different ways to use more than one command on a single command line:

Operator	Action	Example
; (semicolon)	Causes commands to run in sequence.	*cmd1*;*cmd2*
&&	Runs the next command if current command succeeds.	*cmd1* && *cmd2*
||	Runs the next command if the current command fails.	*cmd1* || *cmd2*
&	Causes command to run in the background. (Described under "Running Background Processes—The & Operator" on page 4-11.)	*cmd1* > *file* & *cmd2* &
|	Creates a pipeline. (Described under "Using Pipes and Filters" on page 5-4.)	ls|wc

Figure 5-2. Multiple Command Operators

Separating Commands on the Same Line with the ; (Semicolon)

You can type more than one command on a line if you separate commands with the ; (semicolon). In the following example, the shell runs `ls` and waits for it to finish:

```
$ ls ; who ; date ; pwd
change
file3
newfile
uname       console/1       Jun 5 14:39
Wed Jun 5  14:42:51  CDT  1985
/u/uname
$ _
```

When `ls` is finished, the shell runs `who`, and so on through the last command.

To make the command line easier to read, you can separate commands from the ; (semicolon) with blanks or tabs. The shell ignores blanks and tabs used in this way.

Making Commands Conditional—The || and && Operators

When you connect commands with the || or && operators, the shell runs the first command and then runs the remaining commands only under the following conditions:

`||`	The shell runs the next command if the current command does not complete (that is, if the command fails [returns a nonzero value]).
`&&`	The shell runs the next command if the current command completes (that is, the command succeeds [returns a value of zero]).

In the following example, the shell checks the exit status of *cmd1*:

```
$ cmd1 || cmd2
$ _
```

If *cmd1* fails, the shell runs *cmd2*. (If *cmd1* succeeds, the shell abandons the command line and prompts you for another command.)

In the next example, the shell again checks the exit status of *cmd1*:

```
$ cmd1 && cmd2 && cmd3 && cmd4 && cmd5
```

If *cmd1* succeeds, the shell runs *cmd2*. If *cmd2* succeeds, the shell runs *cmd3*, and on through the series until a command fails or the last command ends. (If any command on the command line fails, the shell abandons the command line and prompts you for another command.)

Grouping Commands

The shell provides two ways to group commands:

Command Grouping Symbol	Action
( ) (parentheses)	The shell creates a subshell to run the grouped commands as a separate process.
{ } (braces)	The shell runs the grouped commands as a unit.

Figure 5-3. Command Grouping Symbols

Using ( ) (Parentheses)

In the following example, the shell runs the commands enclosed in ( ) (parentheses) as a separate process:

```
$ (cd x;ls);ls
$ _
```

The shell creates a ***subshell*** (a separate shell program) that moves to directory x (`cd x`) and lists the files in that directory (`ls`). The first shell does not change directories. After the subshell process is complete, the shell lists the files in the current directory (`ls`).

If this command were written without the ( ), the original shell would move to directory x, list the files in that directory, and then list the files in that directory again. There would be no subshell and no separate process for the `cd x;ls` command.

The shell recognizes the ( ) wherever they occur in the command line. To use parentheses literally (that is, without their command-grouping action), quote them by placing a \ (backslash) immediately before the (or), for example, \(. (For more information on quoting in the shell, see "Quoting" on page 5-11.)

Using { } (Braces)

When commands are grouped in { }, the shell executes them without creating a subshell. In the following example, the shell runs `date` and writes its output to the file `today.grp`, then runs `who` and writes its output to `today.grp`:

```
$ { date; who; } >today.grp
$ _
```

If the commands were not grouped together with braces, the shell would write the output of `date` to the display and the output of `who` to the file.

The shell recognizes { } (braces) in pipelines and command lists, but only if the left brace is the first character on a command line. (For other meanings of braces in the shell, see "Using Braces as Delimiters" on page 6-6.)

Quoting

Reserved characters are characters such as < > | & ? and * that have a special meaning to the shell. "Shell Reserved Characters and Words" on page 6-34 lists all the shell reserved characters. To use a reserved character literally (that is, without its special meaning), quote it with one of the three shell quoting conventions:

Quoting Convention	Action
\	(backslash) Quotes a single character.
' '	(single quotes) Quote a string of characters (except the ' itself).
" "	(double quotes) Quotes a string of characters (except $, `, and \).

Figure 5-4. Shell Quoting Conventions

Using the Backslash

To quote a single character, place a \ (backslash) immediately before that character:

```
$ echo \?
?
$ _
```

This command returns a single ? character.

Using ' ' (Single Quotes)

When you enclose a string of characters in single quotes, the shell takes every character in the string (except the ' itself) literally.

The following example shows how single quotes can be used in the arguments for a command:

```
$ echo x'>'y+0
x>y+0
$ _
```

The echo command returns the string x>y+0 because the single quotes remove the special meaning of the > reserved character.

You also can use single quotes when you assign values to variables. For more information about using single quotes with variables, see "Single Quotes in Variable Assignments" on page 6-7.

Using " " (Double Quotes)

Double quotes provide a special form of quoting. Within double quotes, the reserved characters $, ` (grave accent), and \ keep their special meanings. The shell takes literally all other characters within the double quotes. Double quotes are most frequently used in variable assignments. For more information about using double quotes with variables, see "Double Quotes in Variable Assignments" on page 6-7.

Matching Patterns

The shell gives you five different ways to match character patterns:

Pattern-Matching Character	Action	Example
*	Matches any string, including the null string.	th* matches th, theodore, and thermohaline.
?	Matches any single character.	304?b matches 304Tb, 3045b, 304Bb, or any other string that begins with 304, ends with b, and has one character in between.
[...]	Matches any one of the enclosed characters.	[A G X]* matches all file names in the current directory that begin with A, G, or X.
[.-.]	Matches any character between the enclosed pair, including the pair.	[T-W]* matches all file names in the current directory that begin with T, U, V, or W.
[!...]	Matches any single character except one of those enclosed.	[!abyz]* matches all file names in the current directory that begin with any character except a, b, y, or z.

Figure 5-5. Shell Pattern-Matching Characters

Naming Files with Pattern-Matching

Commands often take file names as arguments. To use several different file names as arguments to a command, you can type out the full name of each file, as the next example shows:

```
$ wc first.t second.t third.t fourth.t fifth.t
$ _
```

However, if the file names have a common pattern (in this example, the `.t` suffix), the shell can match that pattern, generate a list of those names, and automatically pass them to the command as arguments.

The * matches any string of characters. In the following example, the name of every text file in this directory includes the suffix `.t`. (This directory contains the same five files shown in the previous example: `first.t . . . fifth.t`.)

```
$ wc *.t
```

The `*.t` matches any file name that begins with a string and ends with `.t`. The shell passes every file name that matches this pattern as an argument for **wc**.

Thus, you do not have to type (or even remember) the full name of each file in order to use it as an argument. Both commands (**wc** with all file names typed out, and **wc *.t**) do the same thing—they pass all files with the `.t` suffix in the directory as arguments to **wc**.

Note: There is one exception to the general rules for pattern-matching. When the first character of a file name is a period, you must match the period explicitly. For example, `echo *` displays the names of all files in the current directory that do not begin with a period. The command `echo .*` prints all file names that begin with a period. This restriction prevents the shell from automatically matching the relative directory names `.` ("dot," which stands for the current directory) and `..` ("dot dot," which stands for the parent directory). For an explanation of relative directory names, see "Using Relative Directory Names (. and .. Notation)" on page 3-18.

If a pattern does not match any file names, the shell returns the pattern itself as the result of the matching operation. For example, if the current directory does not contain any file names that end with .c, the command echo *.c returns *.c.

Using echo with Pattern-Matching Characters

You can use the **echo** command to learn how the shell interprets pattern-matching characters. The following example is based on a directory that has the files:

```
part1
part2
part3
pre.txt
post.txt

$ echo *
part1  part2  part3  pre.txt  post.txt
$ echo *.*
pre.txt  post.txt
$ echo part?
part1  part2  part3
$ echo ????.???
post.txt
$ echo *[1357]
part1  part3
$ echo [a-o]*
[a-o]*
$ echo [!abc]*
part1  part2  part3  pre.txt  post.txt
$ _
```

Each **echo** command in the example uses one or more pattern-matching characters in its argument, and returns different information about the files in the directory. Notice that echo [a-o]* does not return any file names because none of the file names in this directory begin with one of the lowercase letters a through o.

Writing and Running Shell Procedures

Besides running commands from the command line, the shell can read and run commands contained in a file. Such a file, called a ***shell procedure*** or ***shell script***, is a program that you can use alone or as a part of a program written in another programming language, such as C, BASIC, or FORTRAN.

Even if you do not usually write programs in other languages, you may find that shell procedures are easy to develop and can make your work on the AIX system more efficient. If you do develop programs routinely, shell procedures provide a quick way to try out program segments before you code and compile them and to build program development tools. In either case, because a shell procedure is an ordinary text file that does not have to be compiled, it is relatively easy to create and maintain.

Note: Some AIX commands or programs are shell procedures. As you become more familiar with writing shell procedures, you may want to study ones supplied with the system for ideas. Look first at **/etc/rc** (the procedure that runs automatically when you start the system), and at any of the files containing shell commands that are located in **/bin**, **/usr/bin**, and **/usr/lib/acct**.

To Write and Run a Shell Procedure

1. Use a text editor to create a file of shell and AIX Operating System commands.
2. Use the **chmod** command to give the file **x** (execute) status.

Writing a Shell Procedure

The first step in writing a shell procedure is to create a file of the commands you need to accomplish a task. Create this file as you would any text file—with **ed** or another editing program. Shell procedures can contain any system command (described in *AIX Operating System Commands Reference*) or shell command (described under "Shell Control Commands" on page 6-22 and **sh** in *AIX Operating System Commands Reference*).

Running a Shell Procedure

If you will use the procedure regularly, you should use the **chmod** command to give it **x** (execute) status. For example, the command `chmod g+x reserve` gives execute status to the file named `reserve` for any user in the group (**g**). (For more details on the **chmod** command, see "Changing Permissions—The chmod (Change Mode) Command" on page 3-45 or **chmod** in *AIX Operating System Commands Reference.*) After you give the file **x** status, run the procedure by simply entering its name (or path name if the procedure file is not in your current directory).

If you will use the procedure only a few times and then discard it, you do not have to give it execute status. However, to run a procedure that does not have execute status, you first must run the shell command (**sh**). For example, if the procedure `reserve` does not have execute status, use the command `sh reserve` to run it. (If the procedure file is not in your current directory, use its path name rather than its simple file name.)

Creating a Shell Procedure—Example

The next example shows every step required to create the simple shell procedure named `lss`:

```
$ ed
a
# lss: list, sorting by size
ls -s | sort
.
w lss
q
$ chmod +x lss
$ _
```

Following is an explanation of each step in the creation of `lss`:

`ed`

Starts the **ed** line editor.

`a`

Causes **ed** to add text to the buffer.

`#: list, sorting by size`

Comment line describing the purpose of the procedure.

`ls -s | sort`

Enters the text (commands) of the procedure itself.

`.`

(period) Stops the editor from adding text to the buffer. The period must be entered in the first position on a line by itself.

`w lss`

Writes (copies) the text from the buffer into the file `lss`.

```
q
```

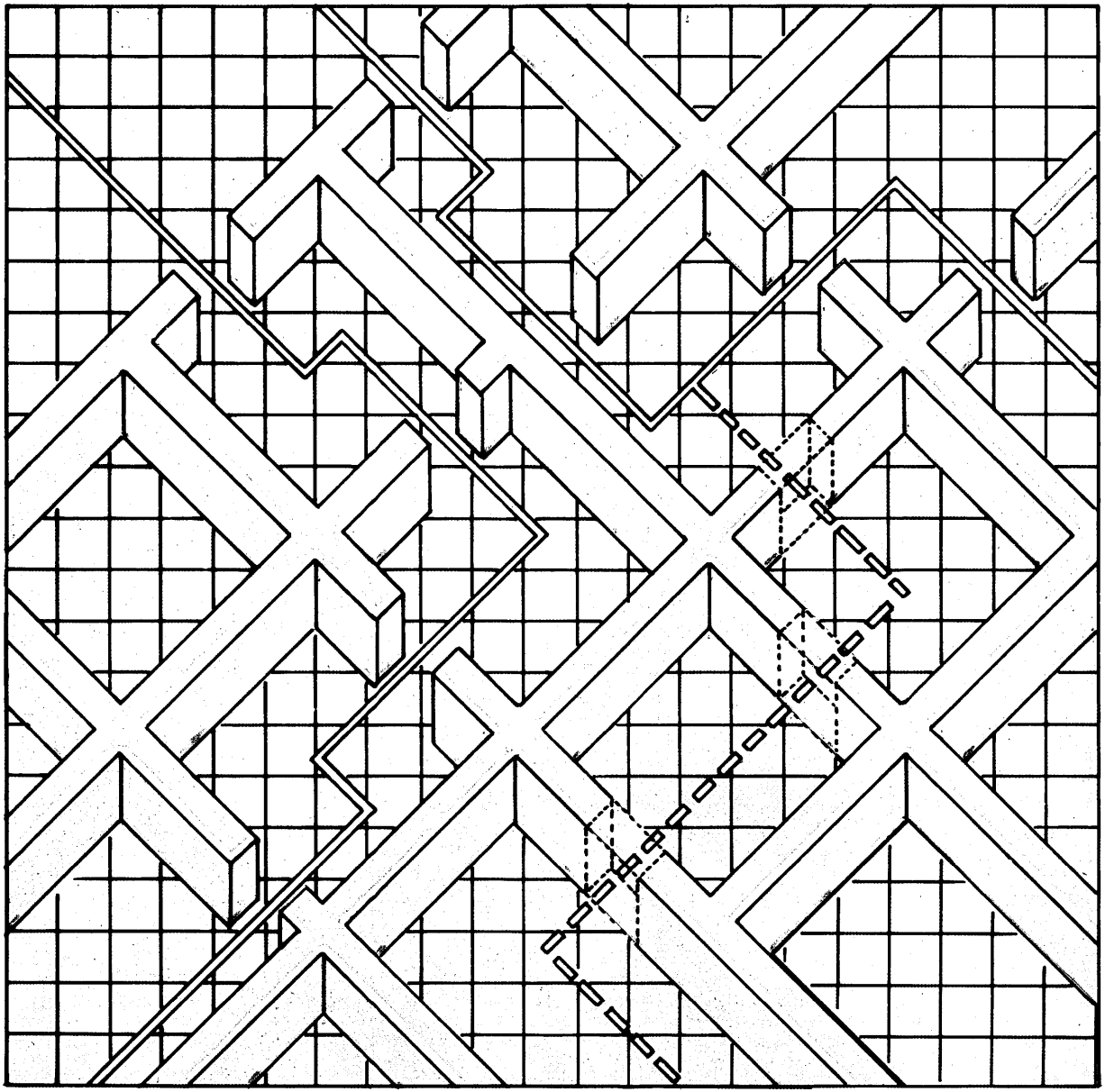
Quits (ends) the editing session.

```
chmod +x lss
```

Gives execute status (**+x**) to the file `lss` for all classes of users.

The `lss` procedure first finds the size, in blocks, for each entry in a directory (`ls -s`). Output from the **ls** command is then piped to the **sort** command (`| sort`). The `sort` command then arranges its standard input according to size, and writes the size and name of each file to standard output. To run the `lss` procedure, simply enter `lss`. Shell procedures are especially useful for routine tasks, allowing you to use multiple commands by entering a single name.

Chapter 6. Using Advanced Shell Features—A Reference



CONTENTS

About This Chapter

This chapter explains the advanced features of the shell. Many of the features covered in this chapter can be used either on the command line (to affect how an individual command runs) or in shell procedures (programs).

Please note two important differences between this chapter and the previous ones:

- This chapter is not for the novice computer user. Unless you have experience with computer programming, you probably should skip this chapter until you are thoroughly familiar with both the AIX system and the earlier chapters of this book.

- This chapter is more like reference material, less like training material. It is organized according to the features of the shell, not according to the jobs you do with those features.

If you work through this chapter from first to last, you should have a general understanding of what you can do with the shell. However, this chapter probably will be most useful when you use it as reference material—when you have a unique job to do and need to understand what shell features can help you do it.

Shell Variables

Like variables in other programming languages, ***shell variables*** are names to which you can assign values. For example, you can assign the value `U. S. A.` to the variable `place` with the following statement:

```
$ place='U. S. A.'
$ _
```

From then on, you can use the variable `place` just as you would use its value. To display the value of variable, type a $ (dollar sign) before the name of the variable. In the following example, the **echo** command displays the value of `place`:

```
$ echo $place
U. S. A.
$ _
```

The shell provides two kinds of variables:

- ***User-defined variables*** (names to which you assign a ***character string***—one or more characters—as a value). Generally, user-defined variables can be set on the command line or in a shell procedure.
- ***Positional parameters*** (variables in shell procedures that refer to values on the command line).

User-Defined Variables

A ***user-defined variable*** is a name to which you assign a specific string value (one or more characters). The name is a sequence of 52 ACSII letters, the 10 ASCII digits, the ASCII underscore, and all extended character. The name cannot begin with an ASCII digit. To create a user-defined variable, use an assignment statement of the form *name=value*. Then, to use the value of the variable, type a $ before the name of the variable. You can use user-defined variables both on the command line and in shell procedures. One special type of user-defined variable, the ***keyword arguments***, can

be used only on the command line. (Keyword arguments are explained under "Keyword Arguments" on page 6-8.)

In the following example, the statement `s=stringofletters` creates the variable `s`:

```
$ s=stringofletters
$ echo $s
stringofletters
$ echo s
s
$ _
```

The command `echo $s` returns the value of `s`. The command `echo s` simply returns the character `s` because a $ does not precede the variable name `s`.

One convenient use for shell variables is as a short notation for long path names. For example, if you routinely use files in the directory `$HOME/personal/correspond/from`, you can assign that path name to the variable name `from`:

```
$ from=$HOME/personal/correspond/from
$ _
```

With this value for the variable `from`, you can use `$from` instead of entering the path name, for example:

```
$ cp tom_5_10_85 $from
$ _
```

Multiple Assignments

You can make more than one variable assignment on a single command line:

```
$ t=text  d=data
$ _
```

In the following example, the shell gives `b` the value `abc`:

```
$ a=$b  b=abc
$ _
```

Then, because b already has a value (since the shell makes that assignment first), the assignment a=$b gives a the same value (abc). Even if the value of b changes, the value of a remains abc.

Using Braces as Delimiters

Use braces { } to separate the name of a variable from any characters that follow immediately. If the character immediately following the variable name is a letter, underscore, or digit, braces are required. In the following example, the purpose of the **echo** command is to display the two strings fun and ction without a space between them.

```
$ a='Form follows fun'
$ echo "${a}ction"
Form follows function
$ _
```

The braces indicate that the enclosed a is a variable name and that its value should be followed immediately by the string ction. (Compare this use of braces with the use described under "Using { } (Braces)" on page 5-10.)

Quoting in Variable Assignments

Certain characters (summarized under "Shell Reserved Characters and Words" on page 6-34) have special meanings to the shell. In variable assignments, you may need to use these characters literally—that is, without their special meanings. To use a special character literally, you must ***quote*** it (using the same quoting conventions described under "Quoting" on page 5-11). The shell takes literally all characters enclosed in single quotes (') except for the single quote itself. Within double quotes, shell takes blanks, tabs, semicolons, and new-lines literally, but substitutes the values for variable names.

Note: In variable assignments, you do not need to quote the special pattern-matching characters (* ? [...]), because pattern-matching does not apply in this context.

Single Quotes in Variable Assignments: In the following example, the value assigned to the variable `stuff` is enclosed in single quotes:

```
$ stuff='echo $ ?  $ *; ls * |  wc'
$ _
```

The value of the variable `stuff` is the literal string `echo $ ?  $ *; ls * | wc`. The shell does not run any of the commands (`echo`, `ls`, or `wc`) and does not give the reserved characters (`$ ?  * |`) their special meanings.

Double Quotes in Variable Assignments: Within double quotes ("), the reserved characters $, ` (grave accent), and \ keep their special meanings. The shell takes literally all other characters within the double quotes.

In the following example, the first line of assignments gives values to the variables `h`, `o`, `c`, and `e`, and the second line gives a value to the variable `e`:

```
$ h=hydrogen  o=oxygen  c=carbon
$ e="Three elements:  $h; $o; $c"
$ echo $e
Three elements:  hydrogen; oxygen; carbon
$ _
```

Because the value of `e` is enclosed in double quotes, the shell takes literally the blanks and semicolons in the string. At the same time, the $ keeps its special meaning, and causes the shell to substitute the values of the variables `h`, `o`, and `c`. Thus, the command `echo $e` returns the value of `e` with the substituted values of `h`, `o`, and `c`.

To quote the $, `, or \ characters within double quotes, place a \ (backslash) immediately before the character. See "Command Substitution" on page 6-13 for an explanation of grave accents (`) and how they work in quoted strings.

Variable Substitution and Quoted Blanks: Ordinarily, the shell interprets blanks between words on a command line as delimiters (separators) between a command and its arguments (and between the arguments themselves). However, after the shell substitutes the value of a variable, it still takes quoted blanks literally (that is, blanks are not reinterpreted as delimiters, even though the string is no longer enclosed in quotes). For example, the following lines assign the same value to `$first` and `$second`:

```
$ first='a string with embedded blanks'
$ second=$first
$ _
```

Compare the assignment in this example with the assignment `first=a string with embedded blanks` (the same string without the quotes). The shell would read `first=a` as a keyword argument (see "Keyword Arguments"), `string` as the command name, and `with`, `embedded`, and `blanks` as arguments to `string`.

Keyword Arguments

The variables that affect a command are called the ***environment*** of the command. The environment can include variables called ***keyword arguments***—variables you set on the command line when you enter the command. In the following example, the keyword argument assigns the value `fred` to the variable name `user` and that assignment becomes part of the environment of the `echo` command:

```
$ user=fred echo $user
fred
$ _
```

Note: The variable assignment in this example affects only the environment of the command, not the environment of the shell from which the command is run.

Keyword arguments usually precede the command name. However, if you set the **-k** flag, the shell takes all arguments of the form *variable*=*value* as keyword arguments:

```
$ set -k usr=fred command black=green tree=frog
$ _
```

(For more information about using keyword arguments, see "The export Command" on page 6-14. For more information about shell flags, see "Shell Flags" on page 6-31.)

Positional Parameters

A ***positional parameter*** is a name that refers to a string in a particular position on the command line. The positional parameter $0 refers to the first string on the command line (usually the name of a command), $1 refers to the second string (the first argument to the command), and so on. When you run a command, the shell creates a positional parameter for each string on the command line up to $9.

Positional parameters allow a shell procedure to take information from the command line (for example, to assign a value to a variable each time the procedure runs). For example, in the following procedure, the **echo** command displays the second argument on the command line (the value of positional parameter $3):

Note: The line that begins with the # character is a comment, not a functional part of the procedure.

```
# posparm3 procedure--demonstrate how pos. parameters work
echo $3
```

Note: To follow this example on your system, first use an editor to create the posparm3 file and then give the file execute status with the **chmod** command (chmod +x posparm3).

You can enter posparm3 with more than two arguments (character strings). The procedure returns only the value of positional parameter $3. In the following example, the posparm3 procedure has five arguments:

```
$ posparm3 Bob Jones Dept. 546 Accounting
Dept.
$ _
```

Each time you run this procedure, you can use different arguments.

The shell automatically creates positional parameters for arguments in positions up to $9. To use arguments in positions numbered higher than 9, you can use the $* notation described under "The shift Command" on page 6-15, or a **for** loop, described in "for—Looping Over a List" on page 6-25.

How the Shell Uses Variables

"Shell Variables" on page 6-4 describes the different types of shell variables. This section explains, first, how the shell ordinarily uses variables and, second, how you can control the way the shell treats variables.

Parameter Substitution

As is explained under "User-Defined Variables" on page 6-4, the shell substitutes the value of a variable (or parameter) for the name of the variable when you type a $ before the name. In the following example, `echo` returns the value of variable `p`:

```
$ p=45ABA54
$ echo $p
45ABA54
$ _
```

If a variable is not set (does not have a value assigned), the shell ordinarily substitutes the null string for the name of the variable. (For example, if `q` does not have a value, then `echo $q` returns nothing to the display.) However, with the notation `${variable-string}`, you can cause the shell to return a string (rather than the null) when a variable does not have a value to substitute. In the following example, variable `q` does not have a value:

```
$ echo ${q-x}
x
$ _
```

Since `q` does not have a value, `echo` returns `x` (the default string).

If you use any special characters in a default string, quote them with the usual shell quoting conventions. In the following example, `echo` returns * if variable `q` is not set:

```
$ echo ${q-'*'}
*
$ _
```

You also can use the value of another variable as a default string. For example, the following echo command returns the value of $1 if q is not set:

```
$ echo ${q-$1}
$ _
```

If $1 is not set, the shell returns the null string.

The notation ${variable=string} actually assigns a value to a variable that is not set. In the following example, if q is not set, the shell assigns it the value 12B1:

```
$ echo ${q=12B1}
$ _
```

Note: The ${variable=string} notation does not work for positional parameters.

If an appropriate default string does not exist, you can use the following notation:

```
$ echo ${q?message}
12B1
$ _
```

The shell substitutes the value of q if it has one. If q does not have a value, the shell returns the word message and ends the procedure.

If a procedure can run only with certain parameters set, you can use the : command to determine whether those parameters have values. The following line begins a procedure that must have three variables set before it can run:

```
:${user?} ${acct?} ${bin?}
```

The : command does nothing once the shell evaluates its parameters. If any of its parameters (user, acct, and bin in this example) are not set, the : command causes the shell to abandon the procedure.

Command Substitution

When you enclose a command in ` ` (grave accents), the shell replaces the name of the command with the standard output of that command. For example, if the current directory is /u/fred/bin, then the following assignments are equivalent:

```
d=`pwd`

d=/u/fred/bin
```

Within ` ` (grave accents), the shell quoting conventions explained under "Quoting" on page 5-11 apply, with one exception: you must use the \ (backslash) to quote a ` (grave accent).

When ` ` (grave accents) appear in strings that are enclosed in double quotes (" . . . ` command name ` . . ."), they are not quoted, as is explained under "Double Quotes in Variable Assignments" on page 6-7). That is, the shell reads them as command substitution reserved characters. In the following example, double quotes cause the shell to take literally the blanks and ? character in the value assigned to the variable where:

```
$ where="Where am I?  `pwd`"
$ echo $where
Where am I?  usr/fred/bin
$ _
```

The shell substitutes the standard output of pwd (the current directory) for `pwd`.

Within double quotes, a command or reserved character enclosed in ` ` (grave accents) retains its special meaning, even though it is part of a quoted string. When grave accents appear in strings enclosed in single quotes (' . . . ` command name ` . . . '), the shell takes literally the grave accents and any commands or reserved characters they enclose. (For more information on using single quotes, see "Single Quotes in Variable Assignments" on page 6-7.)

The export Command

With the **export** command, you can set user-defined variables to apply to any subsequent commands (rather than setting the keyword argument for each command when you enter it). This process, called ***marking the arguments for export***, is useful if, for example, you want the same user-defined variables to be part of the environment for several different commands. In the following example, after values are assigned to variable names `user` and `box`, the `export` command marks the variables `user` and `box` for export:

```
$ user=jason box=square
$ export user box
$ echo $user $box
jason square
$ _
```

The subsequent `echo` command displays the values of `user` and `box`.

To get a list of the variables currently marked for export, enter `export`.

If you want the value of a variable to remain constant, declare it ***readonly***. In the following example, the variables `user` and `box` are declared `readonly`:

```
$ readonly user box
$ _
```

After this declaration, the values of `user` and `box` cannot be changed by a subsequent variable assignment.

To get a list of all variables declared **readonly**, simply enter `readonly`. Once you declare a variable readonly, you can change its value only by deleting the variable and then re-creating it. To delete the variable, use the shell **unset** command (described under **sh** in *AIX Operating System Commands Reference*); if you do not delete the variable, it is automatically deleted when you log out.

The shift Command

The **shift** command, used with positional parameters, shifts arguments to the left one string at a time. For example, the **shift** command discards the value of positional parameter $1, replaces $1 with $2, replaces $2 with $3, and so on. The **shift** command does not shift the $0 positional parameter (the name of the procedure). With each shift, the positional parameter with the highest number becomes ***unset*** (that is, there is no longer an argument in that position). An argument can cause the **shift** command to shift more than one string at a time (for example, `shift 2` causes **shift** to move two strings at a time).

To demonstrate the **shift** command, the following procedure uses three shell features that have not yet been introduced in this book. The **while** statement means *as long as a specified condition exists.* The **test** command determines whether or not a condition exists (in the example, the command `test $# != 0` means *test for a positional parameter that is not 0.* The control command pair **do** and **done** create a ***loop***—a series of commands to be repeated until a specified condition changes; in the example, the commands are **echo**, which displays the values of all the positional parameters, and **shift**, which shifts the positional parameters to the left one at a time. Thus, the procedure displays the values of the positional parameters, shifts the arguments to the left, displays their values again, and so on, continuing until only the $0 positional parameter (the one that refers to **echo**) remains. (The line that begins with the # character is a comment, not a functional part of the procedure.)

Note: To follow this example on your system, first use an editor to create the file `ripple` and then give the file execute status with the **chmod** command (for example, `chmod +x ripple`).

```
#          ripple command
while test $# != 0
do
          echo $1  $2  $3  $4  $5  $6  $7  $8  $9
          shift
done
```

To run the ripple command, enter ripple *string1 string2 string3 . . . :*

```
$ ripple 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
2 3 4 5 6 7 8 9
3 4 5 6 7 8 9
4 5 6 7 8 9
5 6 7 8 9
6 7 8 9
7 8 9
8 9
9
$ _
```

At most, the echo command in ripple displays the values of nine arguments.

To pass more than nine arguments from the command line to a procedure, use the $* notation. For example, in the ripple procedure, the echo command could be written echo $*:

```
#           ripple command
while test $# != 0
do
            echo $*
            shift
done
```

If there are nine or fewer arguments, these two versions of the **echo** command produce the same result. However, if there are more than nine arguments, only the command echo $* accesses all of them.

You can also use a **for** loop to pass additional arguments to the shell. See "for—Looping Over a List" on page 6-25 for more information on using a **for** loop.

The set Command

The shell automatically assigns positional parameters from command line arguments. However, you can assign values to positional parameters from within shell procedures with the **set** command.

In the following example, the **set** command is one line from a shell procedure:

```
set   abc  def  ghi
```

This **set** command first makes these assignments:

```
abc  =  $1
def  =  $2
ghi  =  $3
```

Second, it unsets (clears) the remaining positional parameters (from $4 on), even if they were set before (for example, if there were arguments to the invoking command). The **set** command cannot assign a value to $0, which always refers to the name of the shell procedure.

The read Command

Like the ` (grave accents) used in command substitution, the **read** command lets you assign values to variables indirectly. The **read** command takes a line from its standard input (usually the keyboard) and assigns words from that line, one by one, to named variables. In the following example, the **read** command assigns words to three named variables, `first`, `init`, and `last`:

```
read first init last
```

With the input line `B. T. Andover`, the **read** command produces the same results as would the following variable assignments:

```
first=B. init=T. last=Andover
```

If there are excess words on the input line (that is, if there are more words than there are variables), they are assigned to the last variable.

Special Shell Variables

The shell program uses several special variables. The shell sets some of these variables, and you can set or reset all of them. Following is a partial list of the special variables and a brief description of how the shell uses each one. (For a complete list of the shell variables, see **sh** in *AIX Operating System Commands Reference.*)

MAIL The path name of the file where your mail is deposited. You must set **MAIL**, and this is usually done in the file **.profile** in your login directory. (When you login, you receive an announcement of any mail in your standard mail file, whether **MAIL** is or is not set.)

HOME The name of your login directory. If you use the **cd** (change directory) command without arguments, **cd** changes the current directory to the value of **HOME**. (In shell procedures, you can use **HOME** to avoid having to use full path names—something that is especially helpful if the path name of your login directory changes.) **HOME** is set by the **login** command.

PATH A list of directories that contain commands. When the shell runs a command, it searches a list of directories for a file of that name that can be run. If **PATH** is not set, the shell searches the current directory, `/bin`, and `/usr/bin`. When **PATH** is set, its value is an ordered list of directory names separated by colons, as is shown in the following example:

```
PATH=:/u/fred/bin:/bin:/usr/bin
```

This **PATH** tells the shell to search the current directory (specified by the null string before the first : [colon]), `/u/fred/bin`, `/bin`, and `/usr/bin`, in that order. Thus, the **PATH** variable lets you have a personal directory of commands that you can access regardless of your current directory. Usually, **PATH** is set in the **.profile** file.

NLCTAB The variable that specifies the path name of the file containing the current collating table. If this is not specified, and a definition is contained in **NLFILE**, the definition **NLFILE** is used. Otherwise, the path nameused is **etc/nls/ctab/default**.

NLFILE The variable that specifies the path name of a text file containing configuration information that describes extended character configuration information. For more information, see Overview of International Character Support in *Managing the AIX Operating System*

CDPATH The variable that tells the shell where to search for the argument to a **cd** command whenever that argument is not null and does not begin with /, ., or ... The value of **CDPATH** is an ordered list of directory path names separated by colons.

A null character anywhere in the list represents the current directory. If the list begins with a colon, a null character is assumed to be before the colon. Initially, **CDPATH** is not set, which means that the shell searches only the current directory. In the following example, the **cd** command is set to search the current directory first, and then to search the home directory:

```
CDPATH=:$HOME
```

Usually **CDPATH** is set in your **.profile** file. If the **cd** command changes to a directory that is not a descendent of the current directory, the shell writes the full path name of the new directory on the diagnostic output.

PS1 The variable that specifies the primary prompt string—the string that the shell displays when it is ready to accept a command. The standard primary prompt string is $ (a dollar sign followed by a blank). PS1 is usually set in the file `$HOME/.profile`. If PS1 is not set, shell uses the standard primary prompt string. To change the primary prompt string, edit `$HOME/.profile` and either:

- Change the value of PS1 (if it is set), or

- Add the line `PS1=`*string* (where string is the primary prompt string you choose).

PS2 The variable that specifies the secondary prompt string—the string that the shell displays when it requires more input after a new-line (that is, after you press **Enter**). The standard secondary prompt string is > (a > symbol followed by a blank). PS2 is usually set in the file `$HOME/.profile`. If PS2 is not set, shell uses the standard secondary prompt string. To change the secondary prompt string, edit `$HOME/.profile` and either: (1) change the value of PS2 or (2) add the line `PS2=`*string*.

IFS The variable that specifies what characters can be used as internal field separators (IFS); these are the characters the shell uses during blank interpretation. The shell initially sets **IFS** to include the blank, tab, and new-line characters.

? The exit status (return code) of the last command run, given as a decimal string. Most commands return a zero exit status if they complete successfully, and a nonzero exit status otherwise.

The number of positional parameters, given as a decimal string.

$ The process number of the current shell, given as a decimal string. All existing processes have unique process numbers.

! The process number of the last process run in the background, given as a decimal string.

- The current shell flags.

Shell Control Commands

Often, you may want a shell procedure to do one thing under certain conditions and another thing when those conditions change (or are never met). For example, you may want part 1 of a procedure to run if the procedure receives `yes` as input, and part 2 if the procedure receives `no` as input. With input `yes`, control of the procedure goes to part 1. With input `no`, control goes to part 2.

The shell provides the following six ***control commands***, or command pairs, that let you pass control to various parts of a procedure or control how a procedure ends:

- **break** and **continue** (loop control)
- **case** (multiway branch)
- **exit** and **trap** (process ending control)
- **for** (looping over a list)
- **if** (structured conditional branch)
- **while** and **until** (conditional looping)

This section lists the control commands in alphabetical order.

break and continue—Loop Control

The **break** command ends a **while**, **until**, or **for** loop. The **continue** command starts the next instance of a loop. Both **break** and **continue** work only when they are used between **do** and **done**.

In the following example, the **case** command compares input from the keyboard (entered in response to the prompt `Please enter data`) with the three strings, `"did"`, `" "`, and `*`. (The lines that begin with # are comments, not functional parts of the procedure.)

```
#This procedure is interactive; 'break' and 'continue'
#commands are used to allow the user to control data entry.
while true
do
        echo "Please enter data"
        read response
        case "$response" in
              "done")    break          #no more data
                         ;;
              " ")       continue
                         ;;
              *)
                         process the data here
                         ;;
        esac
done
```

If the entered data match done, the **break** command ends the loop and causes the procedure to start again after done (the end of the enclosing loop). If the entered data match " " (a space), the **continue** command causes the procedure to start again at the **while** (or **until** or **for**) that begins this enclosing loop. If the entered data match *, the procedure processes the data and then completes normally (**esac** ends the **case** statement).

The **break** command exits from the innermost enclosing loop and causes the procedure to start again after the next (unmatched) **done**. To restart the procedure more than one level up from the loop containing **break**, use break *n*, where *n* specifies the number of levels.

The **continue** command causes the procedure to start again at the nearest enclosing **while**, **until**, or **for** (that is, the one beginning the innermost loop containing the **continue**). To restart at any loop other than the innermost enclosing one, use continue *n*, where *n* specifies how many loops (levels) up the **continue** is to operate.

case—The Multiway Branch

With the **case** command, you can create multiway branches. The following example shows the general format for the **case** command:

```
case string in
   pattern) command list;;
   .
   .
   .
   pattern) command list;;
esac
```

The shell attempts to match `string` with each `pattern` in turn. When the shell finds a `pattern` that matches `string`, it runs the command list following that `pattern`. The shell matches only one pattern (that is, if more than one `pattern` in the list matches `string`, the shell runs the command list after the first matching pattern, and then ends the `case` command.

The `;;` (double semicolon) symbol causes the shell to break out of the **case** procedure. It is required after all but the last command list. You can use the standard shell pattern-matching characters with **case**. (Pattern-matching characters are described under "Matching Patterns" on page 5-13.)

The exit and trap Commands

With the **exit** and **trap** commands, you can control the way a process ends (terminates). The **exit** command ends a process before the process reaches end-of-file. If **exit** has an argument, it sets the exit status of the process to the value of that argument. For example, the command `exit 0` in a procedure causes the exit status of that procedure to be 0 (that is, successful). If the argument is omitted, **exit** uses the exit status of the last command that ran.

Ordinarily, an interrupt signal from the keyboard ends a shell procedure. The **trap** command can be set to do any routine tasks necessary to make the termination of a process orderly. In the following example, the **trap** command is set to remove temporary files when signal 2 (interrupt from the keyboard) is received:

```
trap 'rm /tmp/ps$ $; exit' 2
```

The **exit** command is required. Without the **exit**, the procedure would start again at the point where the interrupt was received.

for—Looping Over a List

With the **for** command, you can perform an operation for each of several files, or run a command for each of several arguments. The next example shows the general format of a shell **for** command:

```
for variable in word list
do
        command list
done
```

A word list is a series of strings separated by blanks. The shell runs commands in the command list once for each word in the word list; `variable` takes each word in the word list in turn as its value.

In the following **for** command, the shell runs **wc** for each of the files `top`, `middle`, and `bottom`:

```
for counts in top middle bottom
do
    wc $counts >> countfile
done
```

Each time **wc** runs, its output is directed to a file named `countfile`.

After it is evaluated the first time, the word list is fixed. The **for** command ends when there are no more words in the word list.

You can use the **for** command without the `in word list` statement. In this case, the current positional parameters are used instead of the word list. This feature is convenient if you need to write a command that performs the same command list for an unknown number of arguments.

if—The Structured Conditional Branch

The **if** command can be used with or without an **else** clause. The following example includes an **else** clause:

```
if command list 1
then
      command list 2
else
      command list 3
fi
```

In this example, the shell runs *command list 1.* If the exit status of *command list 1* is zero, the shell runs *command list 2.* If the exit status of *command list 1* is not zero, the shell runs *command list 3.* The word **fi** marks the end of the **if** command.

while and until—Conditional Looping

Following is an example of the general format for the **while** command:

```
while command list 1
do
         command list 2
done
```

The shell runs *command list 1.* If *command list 1* is successful, the shell runs *command list 2.* The shell repeats this sequence until *command list 1* is not successful, and then ends the loop.

The **until** command causes the shell to run a loop as long as the first command list is **not** successful (that is, **until** and **while** test for opposite conditions). In the next example, the shell runs *command list 1* and then checks its exit status (successful or unsuccessful):

```
until command list 1
do
         command list2
done
```

If *command list 1* is not successful, the shell runs *command list 2.* The shell repeats this sequence until *command list 1* is is successful, and then ends the loop.

Inline Input (Here) Documents

The shell commands and procedures usually read their input from the keyboard or from a file (as is explained under "Redirecting Input and Output" on page 4-9). A command in a procedure also can read its input from data contained in the procedure file (***inline input***). The inline input portion of a procedure file is often called a ***here*** document.

A here document begins with the << symbol and ends with a specified string, as the following example shows:

```
for i
do
        grep $i <<!
        ted   abc123
        fred  def456
        tom   ghi789
        sam   jkl198
        frank mn0765
        bill  pqr432
!
done
```

The string that follows << (in this example, !) appears on a line by itself, it marks the end of the here document. In this example, the shell takes the data between <<! and ! as the standard input for **grep**.

The shell substitutes the values of any variables or parameters in the here document before it makes the data available as input to a command. To prevent the shell from substituting the value of specific variables, quote the $ reserved character with a \. To prevent the shell from substituting the values of all variables in a here document, quote the end marking string (! in the previous example) with a \ (backslash).

Standard Error and Other Output

Generally, when a command starts, three files already are open: ***standard input***, ***standard output***, and ***standard error***. If you want to redirect standard input or standard output (for example, cause a command to take its input from a file rather than from the keyboard), you can use the procedures explained under "Redirecting Input and Output" on page 4-9. However, if you want to redirect standard error (or other) output, you must use the methods explained in this section; the methods in this section also can be used to redirect standard input and standard output.

A number, called a ***file descriptor***, is associated with each of the files a command ordinarily uses:

File	File Descriptor	Device
standard input	0	keyboard
standard output	1	display
standard error	2	display

Figure 6-1. Standard File Descriptors

To redirect standard error output, type a 2 (the file descriptor number) before one of the output redirection symbols (> and >>) and a file name after the symbol. For example, the following command adds standard error output from the **cc** command to the file ERRORS:

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

Commands may produce output besides standard and standard error. With the method just described for redirecting standard error output, you can redirect output associated with any file descriptor from 0 through 9. For example, if *cmd* writes output to file descriptor 9, you can redirect that output to the file `savedata` with the following command:

```
$ cmd 9>savedata
```

If a command produces output to several different file descriptors, you can redirect each one independently, as the following example shows:

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

The command *cmd* directs standard output to file descriptor 1, standard error to file descriptor 2, and output for a data file to file descriptor 9.

AIX commands generally use only file descriptors 0, 1, and 2. For more information about using file descriptors to redirect input and output, see **sh** in *AIX Operating System Commands Reference.*

Shell Flags

The shell provides two different types of flags:

- ***Set flags***. These flags, which are put into effect by the ***set*** command, alter the way the shell runs.
- ***Command line flags***. These flags, which are entered on the command line, alter the way the shell starts. Command line flags cannot be set with the **set** command.

Set Flags

To put a set flag into effect, enter `set -`*f* (where *-f* is the name of one or more flags preceded by a hyphen). In the following example, the **set** command turns on the **x** and **v** flags.

```
set -xv
```

To remove a set flag, enter `set +`*flag* (where *+flag* is the name of one or more flags preceded by a plus sign):

```
set +xv
```

Following is a list of set flags that often are useful in shell procedures.

Flag	Explanation
-e	Causes the shell to exit immediately if any command exits with nonzero exit status.
-u	Causes the shell to treat use of an unset variable as an error. This flag can be used to perform a global check on variables.
-t	Causes the shell to exit after reading and running the commands on the remainder of the current input line.

-n Prevents the shell from running commands in a procedure. For example, to check a procedure for syntax errors without running the commands, enter `set -nv` at the beginning of the file.

-k Causes the shell to treat all arguments on the command line of the form *variable=value* as keyword arguments. When **-k** is not set, only arguments of this type that appear before the command name are treated as keyword arguments.

In addition to these set flags, there are two others—the **x** and **v** flags—that are useful for debugging shell procedures. The **x** and **v** flags are usually set from the keyboard.

Flag	Explanation
-x	Causes the shell to print commands and their arguments as they are run. The **-x** flag does not print shell control commands, such as **for**, **while**, **case**, and **if**. **Note:** The **-x** flag traces only the commands that are run, while the **-v** flag causes the shell to print each line of input until a syntax error is found.
-v	Causes the shell to print input lines as they are read. This flag is helpful in finding syntax errors. The commands on each input line are run after that input line is printed, unless the **-n** flag is also in effect.

Command Line Flags

The following list contains descriptions of the four shell command line flags. These flags are specified on the command line and cannot be turned on with the **set** command.

Flag	Explanation
-i	Starts an interactive shell. (If this flag is specified or if both input and output are connected to the display station, the shell is interactive.)
-s	Causes the shell to read commands from standard input. (If this flag is specified, or if input is not redirected, the shell reads commands from standard input.) The shell output is written to file descriptor 2. When you log in to the system, your initial shell operates as if the **-s** flag is turned on.
-c	Causes the shell to read commands from the first string following the flag. Remaining arguments are ignored. Double quotes should be used to enclose a multiword string in order to allow for variable substitution.
-r	Starts the restricted shell. In the restricted shell, certain commands are not available. For example, the **cd** command produces an error message, and you cannot set PATH. For more information on the restricted shell, see **sh** in *AIX Operating System Commands Reference.*

Shell Reserved Characters and Words

Syntactic

|	pipe symbol
&&	'AND' symbol
||	'OR' symbol
;	command separator
;;	case delimiter
&	background commands
( )	command grouping
<	input redirection
< <	input from a here document
>	output creation
> >	output append

Figure 6-2. Shell Reserved Characters and Words—Syntactic

Patterns

*	match any character(s) including none
?	match any single character
[...]	match any of the enclosed characters

Figure 6-3. Shell Reserved Characters and Words—Pattern-Matching

Substitution

${...}	substitute shell variable
` ... `	substitute command output

Figure 6-4. Shell Reserved Characters and Words—Substitution

Quoting

[\]	quote the next character
'...'	quote the enclosed characters except for ' (the single quote itself)
"..."	quote the enclosed characters except for the $, `, [\], and "

Figure 6-5. Shell Reserved Characters and Words—Quoting

Reserved Words

if then else elif fi
case in esac
for while until do done
{ } [ ] test

Figure 6-6. Shell Reserved Characters and Words—Reserved Words

Appendix A. Creating and Editing Files with ed

CONTENTS

About This Chapter

This appendix explains how to create, edit (modify), display, and save text files with **ed**, a line editing program. If your system has another editing program, you may wish to learn how to do these tasks with that program.

A good way to learn how **ed** works is to try the examples in this appendix on your AIX system. Since the examples build upon each other, it is important for you to work through them in sequence. Also, to make what you see on the screen consistent with what you see in this guide, it is important to do the examples just as they are given.

In the examples, everything you should type is shaded in blue (for example, `1,3t5`). When you are told in the text to ***enter*** something, you should type all of the information for that line and then press the **Enter** key.

Note: A line editing program allows you to work with the contents of a file one line at a time. Regardless of what text is on the screen, you can edit only the ***current*** line. If you have experience with a screen editing program, you should pay careful attention to the differences between that program and **ed**. For example, with the **ed** program, you cannot use the **Cursor Up** and **Cursor Down** keys to change your current line.

Understanding Text Files and the Edit Buffer

A ***file*** is a collection of data stored together in the computer under an assigned name. You can think of a file as the computer equivalent of an ordinary file folder—it may contain the text of a letter, a report, or some other document, or the source code for a computer program. File names can be up to 14 characters long and can contain letters, numbers, periods, commas, underscores, and some other characters.

The ***edit buffer*** is a temporary storage area that holds a file while you work with it—the computer equivalent of the top of your desk. When you work with a text file, you place it in the edit buffer, make your changes to the file (edit it), and then transfer (copy) the contents of the buffer to a permanent storage area.

The rest of this appendix explains how to create, display, save, and edit (modify) text files.

Creating and Saving Text Files

To follow this procedure, you must be logged in to your AIX system and have the $ (shell) prompt on your screen.

To Create and Save a Text File

1. At the $ (shell) prompt, enter:

   `ed` *filename*

   Where *filename* is the name of the file you want to create or edit.

2. When you receive the ?*filename* message, enter:

   `a`

3. Enter your text.

4. To stop adding text, enter a . (period) at the start of a new line.

5. Enter:

   `w`

   to copy the contents of the edit buffer into the file *filename.*

6. Enter:

   `q`

   to end the **ed** program.

Starting the ed Program

To start the **ed** program, enter a command of the form `ed` *filename* after the $ (shell) prompt. (In place of *filename*, enter the name you want to assign to the file.)

In the following example, the **ed afile** command starts the **ed** program and indicates that you want to work with a file named **afile**:

Note: If you intend to work through the examples, start with this one.

```
$ ed afile
?afile
_
```

The **ed** program responds with the message `?afile`, which means that the file does not now exist. You can now use the **a** (append) subcommand (described in the next section) to create `afile` and put text into it.

Entering Text—The a (Append) Subcommand

To put text into your file, enter `a`. The **a** subcommand tells **ed** to add, or append, the text you type to the edit buffer. Type your text, pressing **Enter** at the end of each line. When you have entered all of your text, enter a . (period) at the start of a new line.

Note: If you do not press **Enter** at the end of each line, the **ed** program automatically moves your cursor to the next line after you fill a line with characters. However, **ed** treats everything you type before you press **Enter** as one line, regardless of how many lines it takes up on the screen; that is, the line ***wraps around***.

The following example shows how to enter text into the file `afile`:

```
a
The only way to stop
appending is to type a
line that contains only
a period.
.
_
```

If you stop adding text to the buffer and then decide you want to add some more, enter another **a** subcommand. Type the text and then enter a period at the start of a new line to stop adding text to the buffer.

If you make errors as you type your text, you can correct them—before you press **Enter**. Use the **Backspace** key to erase the incorrect character(s). Then type the correct characters in their place.

Displaying Text—The p (Print) Subcommand

Use the **p** (print) subcommand to display the contents of the edit buffer. To display a single line, use the subcommand *n***p** (where *n* is the number of the line):

```
2p
appending is to type a
_
```

To display a series of lines, use the *n*,*m***p** subcommand (where *n* is the starting line number and *m* is the ending line number):

```
1,3p
The only way to stop
appending is to type a
line that contains only
_
```

To display everything from a specific line to the end of the buffer, use the *n*,**$p** subcommand (where *n* is the starting line number and

$ stands for the last line of the buffer). In the following example, 1,$p displays everything in the buffer:

```
1,$p
The only way to stop
appending is to type a
line that contains only
a period.
-
```

Note: Many examples in the rest of this appendix use **1,$p** to display the buffer's contents. In these examples, the **1,$p** subcommand is optional, but convenient—it lets you verify that the subcommands in examples work as they should. Another convenient **ed** convention is **,p**, which is equivalent to **1,$p**—that is, it displays the contents of the buffer.

Saving Text—The w (Write) Subcommand

The **w** (write) subcommand ***writes***, or copies, the contents of the buffer into a file. You can save all or part of a file under its original name or under a different name. In either case, **ed** replaces the original contents of the file you specify with the data copied from the buffer.

Saving Text Under the Same File Name

To save the contents of the buffer under the original name for the file, enter `w`:

```
w
78
-
```

The **ed** program copies the contents of the buffer into the file named **afile** and displays the number of characters copied into the file (78). This number includes blanks and characters such as **Enter** (sometimes called ***newline***) which are not visible on the screen.

The **w** subcommand does not affect the contents of the edit buffer. You can save a copy of the file and then continue to work with the contents of the buffer.

The stored file is not changed until the next time you use **w** to copy the contents of the buffer into it. As a safeguard, it is a good practice to save a file periodically while you work on it. Then, if you make changes (or mistakes) that you do not want to save, you can start over with the most recently saved version of the file.

Note: The **u** (undo) subcommand restores the buffer to the state it was in before it was last modified by an **ed** subcommand. The subcommands that **u** can reverse are: **a**, **c**, **d**, **g**, **G**, **i**, **j**, **m**, **r**, **s**, **t**, **v**, and **V**.

Saving Text Under a Different File Name

Often, you may need more than one copy of the same file. For example, you could have the original text of a letter in two files—one to keep as it is, and the other to be revised.

If you have followed the previous examples, you have a file (named **afile**) that contains the original text of your document. To create another copy of the file (while its contents are still in the buffer), use a subcommand of the form `w` *filename*, as the following example shows:

```
w bfile
78
-
```

At this point, **afile** and **bfile** have the same contents; since each is a copy of the same buffer contents. However, because **afile** and **bfile** are separate files, you can change the contents of one without affecting the contents of the other.

Saving Part of a File

To save part of a file, use a subcommand of the form *n,m***w** *filename*, where:

n is the beginning line number of the part of the file you want to save.

m is the ending line number of the part of the file you want to save (or the number of a single line, if that is all you want to save).

filename is the name of a different file (optional).

In the following example, the **w** subcommand copies lines 1 and 2 from the buffer into a new file named **cfile**:

```
1,2w cfile
44
_
```

Then **ed** displays the number of characters written into **cfile** (44).

Leaving the ed Program—The q (Quit) Subcommand

Warning: You lose the contents of the buffer when you leave the **ed** program. To save a copy of the data in the buffer, use the **w** subcommand to copy the buffer into a file before you leave the **ed** program.

To leave the **ed** program, enter the q (quit) subcommand:

```
q
$ _
```

The **q** subcommand returns you to the **$** (shell) prompt.

If you have changed the buffer, but have not saved a copy of its contents, the **q** subcommand responds with ?, an error message. At that point, you can either save a copy of the buffer (with the **w**

subcommand) or enter q again (which lets you leave the **ed** program without saving a copy of the buffer).

Note: You can log out from the **ed** program by pressing **END OF FILE**. If you have changed the buffer since you last saved it, the system displays the ? error message.

Loading Files into the Edit Buffer

Before you can edit a file, you must load it into the edit buffer. You can load a file either at the time you start the **ed** program or while the program is running.

To Load Files into the Edit Buffer

ed *filename*

This starts **ed** and loads the file *filename* into the edit buffer.

OR

e *filename*

When **ed** is running, this loads the file *filename* into the buffer, erasing any previous contents of the buffer.

OR

*n*r *filename*

When **ed** is running, this reads the named file into the buffer after line *n*. (If you do not specify *n*, **ed** adds the file to the end of the buffer.)

Using the ed (Edit) Command

To load a file into the edit buffer when you start the **ed** program, simply type the name of the file after the command **ed**. The **ed** command in the following example invokes the **ed** program and loads the file `afile` into the edit buffer:

```
$ ed afile
78
-
```

The **ed** program displays the number of characters that it read into the edit buffer (78).

If **ed** cannot find the file, it displays *?filename*. To create that file, use the **a** (append) subcommand (described under "Entering Text—The a (Append) Subcommand" on page A-7) and the **w** (write) subcommand (described under "Saving Text—The w (Write) Subcommand" on page A-9).

Using the e (Edit) Subcommand

Once you start the **ed** program, you can use the **e** (edit) subcommand to load a file into the buffer. The **e** subcommand replaces the contents of the buffer with the new file. (Compare the **e** subcommand with the **r** subcommand, described under "Using the r (Read) Subcommand" on page A-15, which adds the new file to the buffer.)

Warning: When you load a new file into the buffer, the new file replaces the buffer's previous contents. Save a copy of the buffer (with the **w** subcommand) before you read a new file into the buffer.

In the following example, the subcommand `e cfile` reads the file `cfile` into the edit buffer, replacing `afile`:

```
e cfile
44
e afile
78
—
```

The `e afile` subcommand then loads `afile` back into the buffer, deleting `cfile`. The **ed** program returns the number of characters read into the buffer after each **e** subcommand (44 and 78).

If **ed** cannot find the file, it returns **?***filename*. To create that file, use the **a** (append) subcommand, described under "Entering Text—The a (Append) Subcommand" on page A-7, and the **w** (write) subcommand, described under "Saving Text—The w (Write) Subcommand" on page A-9.

You can edit any number of files, one at a time, without leaving the **ed** program. Use the **e** subcommand to load a file into the buffer. After making your changes to the file, use the **w** subcommand to save a copy of the revised file. (See "Saving Text—The w (Write) Subcommand" on page A-9 for information about the **w** subcommand.) Then use the **e** subcommand again to load another file into the buffer.

Using the r (Read) Subcommand

Once you have started the **ed** program, you can use the **r** (read) subcommand to read a file into the buffer. The **r** subcommand adds the contents of the file to the contents of the buffer. The **r** subcommand does not delete the buffer. (Compare the **r** subcommand with the **e** subcommand, described under "Using the e (Edit) Subcommand" on page A-14, which deletes the buffer before it reads in another file.)

With the **r** subcommand, you can read a file into the buffer at a particular place. For example, the `4r cfile` subcommand reads the file `cfile` into the buffer following line 4. The **ed** program then renumbers all of the lines in the buffer. If you do not use a line number, the **r** subcommand adds the new file to the end of the buffer's contents.

The following example shows how to use the **r** subcommand with a line number:

```
1,$p
The only way to stop
appending is to type a
line that contains only
a period.
3 r cfile
44
1,$p
The only way to stop
appending is to type a
line that contains only
The only way to stop
appending is to type a
a period.
_
```

1,$p displays the four lines of afile. Next, the 3 r cfile subcommand loads the contents of cfile into the buffer, following line 3, and shows that it read 44 characters into the buffer. The next 1,$p subcommand displays the buffer's contents again, which lets you verify that the **r** subcommand read **cfile** into the buffer after line 3.

If You Are Working the Examples

If you are working the examples on your AIX system, do the following before you go to the next section:

1. Save the contents of the buffer in the file cfile. Enter:

   ```
   w cfile
   ```

2. Load afile into the buffer. Enter:

   ```
   e afile
   ```

Displaying and Changing the Current Line

The **ed** program is a ***line editor***. This means that **ed** lets you work with the contents of the buffer one line at a time. The line you can work with at any given time is called the ***current line***, and it is represented by the symbol . (called *dot*). To work with different parts of a file, you must change the current line.

To Display the Current Line

To display the current line, enter:

```
p
```

OR

To display the line number of the current line, enter:

```
.=
```

Note: You cannot use the **Cursor Up** and **Cursor Down** keys to change the current line. To change the current line, use the **ed** subcommands described in the following sections.

To Change Your Position in the Buffer

- To set your current line to line number *n*, enter:

  n

- To move the current line forward through the buffer one line at a time:

  Press **Enter**

- To move the current line backward through the buffer one line at a time, enter:

  -

- To move the current line *n* lines forward through the buffer, enter:

  .+*n*

- To move the current line *n* lines backward through the buffer, enter:

  .-*n*

Finding Your Position in the Buffer

When you first load a file into the buffer, the last line of the file is the current line. As you work with the file, you usually change the current line many times. You can display the current line or its line number at any time.

To display the current line, enter p:

```
p
a period.
_
```

The **p** subcommand displays the current line (a period.). Because the current line has not been changed since you read afile into the buffer, the current line is the last line of the buffer.

Enter .= to display the line number of the current line:

```
.=
4
-
```

Since afile has four lines, and the current line is the last line in the buffer, the .= subcommand displays 4.

You also can use the $ (the symbol that stands for the last line in the buffer) with the = subcommand to determine the number of the last line in the buffer:

```
$=
4
-
```

The **$=** subcommand is an easy way to find out how many lines are in the buffer.

Note: The **ed** $ symbol has no relationship to the $ (shell) prompt.

Changing Your Position in the Buffer

You can change your position in the buffer (change your current line) in one of two ways:

- Specify a line number (an absolute position).
- Move forward or backward relative to your current line.

To move the current line to a specific line, enter the line number; **ed** displays the new current line. In the following example, the first line of **afile** becomes the current line:

```
1
The only way to stop
_
```

To move the current line forward through the buffer one line at a time, press **Enter**, as the following example shows:

```
appending is to type a

line that contains only

a period.

?
_
```

Notice that when you try to move beyond the last line of the buffer, **ed** returns ?, an error message. You cannot move beyond the end of the buffer.

To set the current line to the last line of the buffer, enter `$`.

To move the current line backward through the buffer one line at a time, enter - (hyphens) one after the other.

```
-
line that contains only
-
appending is to type a
-
The only way to stop
-
?
_
```

When you try to move beyond the first line in the buffer, you receive the ? message. You cannot move beyond the top of the buffer.

To move the current line forward through the buffer more than one line at a time, enter .*n* (where *n* is the number of lines you want to move):

```
.2
line that contains only
—
```

To move the current line backward through the buffer more than one line at a time, enter: .-*n* (where *n* is the number of lines you want to move):

```
.-2
The only way to stop
—
```

Locating Text

If you do not know the number of the line that contains a particular word or another string of characters, you can locate the line with a ***context search***.

To Make a Context Search

- To search forward, enter:

  /string to find/

- To search backward, enter:

  ?string to find?

Searching Forward Through the Buffer

To search forward through the buffer, enter the string enclosed in // (slashes):

```
/only/
line that contains only
-
```

The context search (**/only/**) begins on the first line after the current line, then locates and displays the next line that contains the string **only**. That line becomes the current line.

If **ed** does not find the string between the first line of the search and the last line of the buffer, then it continues the search at line 1 and searches to the current line. If **ed** searches the entire buffer without finding the string, it displays the ? error message:

```
/random/
?
-
```

Once you have searched for a string, you can search for the same string again by entering //. The following example shows one search for the string only, and then a second search for the same string:

```
/only/
The only way to stop
//
line that contains only
_
```

Searching Backward Through the Buffer

Searching backward through the buffer is much like searching forward, except that you enclose the string in question marks (??):

```
?appending?
appending is to type a
_
```

The context search begins on the first line before the current line and then locates the first line that contains the string appending. That line becomes the current line. If **ed** searches the entire buffer without finding the string, it stops the search at the current line and displays the message ?.

Once you have searched backward for a string, you can search backward for the same string again by entering ??.

Changing the Direction of a Search

You can change the direction of a search for a particular string by using the / and ? search characters alternately:

```
/only/
line that contains only
??
The only way to stop
_
```

If you go too far while searching for a character string, it is convenient to be able to change the direction of your search.

Making Substitutions—The s (Substitute) Subcommand

Use the **s** (substitute) subcommand to replace a ***character string*** (a group of one or more characters) with another. The **s** subcommand works with one or more lines at a time, and is especially useful for correcting typing or spelling errors.

To Make Substitutions

- To substitute *newstring* for *oldstring* at the first occurrence of *oldstring* in the current line, enter:

  s/*oldstring*/*newstring*/

- To substitute *newstring* for *oldstring* at the first occurrence of *oldstring* on line number *n*, enter:

  *n*s/*oldstring*/*newstring*/

- To substitute *newstring* for *oldstring* at the first occurrence of *oldstring* in each of the lines *n* through *m*, enter:

  n,*m*s/*oldstring* /*newstring*/

Substituting on the Current Line

To make a substitution on the current line, first make sure that the line you want to change is the current line. In the following example, the `/appending/` (search) subcommand locates the line to be changed. Then the `s/appending/adding text/p` (substitute) subcommand substitutes the string `adding text` for the string `appending` on the current line. The print (**p**) subcommand displays the changed line.

```
/appending/
appending is to type a
s/appending/adding text/
p
adding text is to type a
-
```

Note: For convenience, you can add the **p** (print) subcommand to the **s** subcommand (for example, `s/appending/adding text/p`). This saves you from having to type a separate **p** subcommand to see the result of the substitution.

A simple **s** subcommand changes only the first occurrence of the string on a given line. To learn how to change all occurrences of a string on the line, see "Changing Every Occurrence of a String" on page A-27.

Substituting on a Specific Line

To make a substitution on a specific line, use a subcommand of the form *n*`s/`*oldstring*`/`*newstring*`/`, where *n* is the number of the line on which the substitution is to be made. In the following example, the **s** subcommand moves to line number 1 and replaces the string `stop` with the string `quit` and displays the new line:

```
1s/stop/quit/p
The only way to quit
-
```

The **s** subcommand changes only the first occurrence of the string on a given line. To learn how to change all occurrences of a string on the line, see "Changing Every Occurrence of a String" on page A-27.

Substituting on Multiple Lines

To make a substitution on multiple lines, use a subcommand of the form *n,m*s/*oldstring* /*newstring*/, where *n* is the first line of the group and *m* is the last. In the following example, the **s** subcommand replaces the first occurrence of the string to with the string TO on every line in the buffer.

```
1,$s/to/TO/
1,$p
The only way TO quit
adding text is TO type a
line that contains only
a period.
-
```

The 1,$p subcommand displays the contents of the buffer, which lets you verify that the substitutions were made.

Changing Every Occurrence of a String

Ordinarily, the **s** (substitute) subcommand changes only the first occurrence of a string on a given line. However, the **g** (global) operator lets you change every occurrence of a string on a line or in a group of lines.

To make a global substitution on a single line, use a subcommand of the form *n*s/*oldstring*/ *newstring*/. In the following example, 3s/on/ON/gp changes each occurrence of the string on to ON in line 3 and displays the new line:

```
3s/on/ON/gp
line that cONtains ONly
-
```

To make a global substitution on multiple lines, specify the group of lines with a subcommand of the form *n,m*s/*oldstring*/ *newstring*/g. In the following example, 1,$s/TO/to/g changes the string TO into the string to in every line in the buffer:

```
1,$s/TO/to/g
1,$p
The only way to quit
adding text is to type a
line that cONtains ONly
a period.
_
```

Removing Characters

You can use the **s** (substitute) subcommand to remove a string of characters (that is, to replace the string with "nothing"). To remove characters, use a subcommand of the form `s/`*oldstring*`//` (with no space between the last two / characters).

In the following example, **ed** removes the string `adding` from line number 2 and then displays the changed line:

```
2s/adding//
text is to type a
_
```

Substituting at Line Beginnings and Ends

Two special characters let you make substitutions at the beginning or end of a line:

Special Substitution Characters

^ (circumflex) Makes a substitution at the beginning of the line. (To get the ^ character, press **Shift-6.**)

$ (dollar sign) Makes a substitution at the end of the line. (In this context, the $ character does not stand for the last line in the buffer.)

To make a substitution at the beginning of a line, use the **s/^/***newstring* subcommand. In the following example, one **s** subcommand adds the string `Remember,` to the start of line number

1. Another **s** subcommand adds the string `adding` to the start of line 2:

```
1s/^/Remember, /p
Remember, The only way to quit
2s/^/adding/p
adding text is to type a
_
```

To make a substitution at the end of a line, use a subcommand of the form `s/$/`*newstring*. In the following example, the **s** subcommand adds the string `Then press Enter.` to the end of line number 4:

```
4s/$/  Then press Enter./p
a period.  Then press Enter.
_
```

Notice that the substituted string includes two blanks before the word `Then` to separate the two sentences.

Using a Context Search

If you do not know the number of the line you want to change, you can locate it with a context search. See "Locating Text" on page A-22 for more information on context searches.

For convenience, you can combine a context search and a substitution into a single subcommand: */*`string to find`*/s/*`oldstring`*/*`newstring`*/*.

In the following example, **ed** locates the line that contains the string `, The` and replaces that string with `, the`:

```
/, The/s/, The/, the/p
Remember, the only way to quit
_
```

Also, you can use the search string as the string to be replaced with a subcommand of the form */*`string to find`*/s//*`newstring`*/*. In the following example, **ed** locates the line that contains the string

cONtains ONly, replaces that string with contains only, and prints the changed line:

```
/cONtains ONly/s//contains only/p
line that contains only
-
```

Deleting Lines—The d (Delete) Subcommand

Use the **d** (delete) subcommand to remove one or more lines from the buffer. The general form of the **d** subcommand is *starting line,ending line*d. After you delete lines, **ed** sets the current line to the first line following the lines that were deleted. If you delete the last line from the buffer, the last remaining line in the buffer becomes the current line. After a deletion, **ed** renumbers the remaining lines in the buffer.

To Delete Lines from the Buffer

- To delete the current line, enter:

  d

- To delete line number *n* from the buffer, enter:

  *n*d

- To delete lines numbered *n* through *m* from the buffer, enter:

  n,*m*d

Deleting the Current Line

If you want to delete the current line, simply enter d. In the following example, the 1,$p subcommand displays the entire contents of the buffer, and the $ subcommand makes the last line of the buffer the current line:

```
1,$p
Remember, the only way to quit
adding is to type a
line that contains only
a period.  Then press Enter.
$
a period.  Then press Enter
d
_
```

The **d** subcommand then deletes the current line (in this case, the last line in the buffer).

Deleting a Specific Line

If you know the number of the line you want to delete, use a subcommand of the form *n*d to make the deletion. In the following example, the 2d subcommand deletes line 2 from the buffer:

```
2d
1,$p
Remember, the only way to quit
line that contains only
_
```

The 1,$p subcommand displays the contents of the buffer, showing that the line was deleted.

Deleting Multiple Lines

To delete a group of lines from the buffer, use a subcommand of the form *n*,*m*d, where *n* is the starting line number and *m* is the ending line number of the group to be deleted.

In the following example, the 1,2d subcommand deletes lines 1 through 2:

```
1,2d
1,$p
?
-
```

The 1,$p subcommand displays the ? message, indicating that the buffer is empty.

If you are following the examples on your AIX system, you should restore the contents of the buffer before you move on to the next section. The following example shows you how to restore the contents of the buffer:

```
e afile
?
e afile
78
-
```

This reads a copy of the original file `afile` into the buffer.

Moving Text—The m (Move) Subcommand

Use the **m** (move) subcommand to move a group of lines from one place to another in the buffer. After a move, the last line moved becomes the current line.

To Move Text

Enter a subcommand of the form *x,ymz* where:

x is the first line of the group to be moved.
y is the last line of the group to be moved.
z is the line the moved lines are to follow.

In the following example, the `1,2m4` subcommand moves the first two lines of the buffer to the position following line 4:

```
1,2m4
1,$p
line that contains only
a period.
The only way to stop
appending is to type a
-
```

The `1,$p` subcommand displays the contents of the buffer, showing that the move is complete.

To move a group of lines to the top of the buffer, use 0 as the line number for the moved lines to follow. In the next example, the `3,4m0` subcommand moves lines 3 through 4 to the top of the buffer:

```
3,4m0
1,$p
The only way to stop
appending is to type a
line that contains only
a period.
-
```

The 1,$p subcommand displays the contents of the buffer, showing that the move has been made.

To move a group of lines to the end of the buffer, use $ as the line number for the moved lines to follow:

```
1,2m$
1,$p
line that contains only
a period.
The only way to stop
appending is to type a
.
```

Changing Lines of Text—The c (Change) Subcommand

Use the **c** (change) subcommand to replace one or more lines with one or more new lines. The **c** subcommand first deletes the line(s) you want to replace and then lets you enter the new lines, just as if you were using the **a** (append) subcommand. When you have entered all of the new text, type a . (period) on a line by itself. The general form of the **c** subcommand is *starting line, ending line*c.

To Change Lines of Text

1. Enter a subcommand of the form:

   n,*m*c

   where:

   n is the number of the first line of the group to be deleted.
   m is the number of the last line of the group (or the only line) to be deleted.

2. Type the new line(s), pressing **Enter** at the end of each line.

3. Enter a period on a line by itself.

Changing a Single Line

To change a single line of text, use only one line number with the **c** (change) subcommand. You can replace the single line with as many new lines as you like.

In the following example, the **2c** subcommand deletes line 2 from the buffer, and then you can enter new text:

```
2c
appending new material is to
use the proper keys to create a
.
1,$p
The only way to stop
appending new material is to
use the proper keys to create a
line that contains only
a period.
-
```

The period on a line by itself stops **ed** from adding text to the buffer. The 1,$p subcommand displays the entire contents of the buffer, showing that the change has been made.

Changing Multiple Lines

To change more than one line of text, give the starting and ending line numbers of the group of lines to be with the **c** subcommand. You can replace the group of lines with one or more new lines.

In the following example, the 2,3c subcommand deletes lines 2 through 3 from the buffer, and then you can enter new text:

```
2,3c
adding text is to type a
.
1,$p
The only way to stop
adding text is to type a
line that contains only
a period.
-
```

The period on a line by itself stops **ed** from adding text to the buffer. The 1,$p subcommand displays the entire contents of the buffer, showing that the change has been made.

Inserting Text—The i (Insert) Subcommand

Use the **i** (insert) subcommand to insert one or more new lines into the buffer. To locate the place in the buffer for the lines to be inserted, you can use either a line number or a context search. The **i** subcommand inserts new lines before the specified line. (Compare the **i** subcommand with the **a** subcommand, explained under "Entering Text—The a (Append) Subcommand" on page A-7, which inserts new lines after the specified line.)

To Insert Text

1. Enter a subcommand of one of the following types:

   *n*i

   Where *n* is the number of the line the new lines will be inserted above.

   /*string*/i

   Where *string* is a group of characters contained in the line the new lines will be inserted above.

2. Enter the new lines.

3. Enter a period at the start of a new line.

Using Line Numbers

If you know the number of the line where you want to insert new lines, you can use an insert subcommand of the form *n*i (where *n* is a line number). The new lines you type go into the buffer before line number *n*. To end the **i** subcommand, type a . (period) on a line by itself.

In the following example, the `1,$p` subcommand prints the contents of the buffer. Then the `4i` subcommand inserts new lines before line number 4.

```
1,$p
The only way to stop
adding text is to type a
line that contains only
a period.
4i
--repeat, only--
.
1,$p
The only way to stop
adding text is to type a
line that contains only
--repeat, only--
a period.
_
```

After `4i` you enter the new line of text and type a period on the next line to end the **i** subcommand. A second `1,$p` subcommand displays the contents of the buffer again, showing that the new text has been inserted.

Using a Context Search

Another way to specify where the **i** subcommand inserts new lines is to use a context search. With a subcommand of the form */string*`/i`, you can locate the line that contains *string* and insert new lines before that line. When you finish inserting new lines, type a period on a line by itself.

In the following example, the `/period/i` subcommand inserts new text before the line that contains the string `period`:

```
/period/i
and in the first position--
.
1,$p
The only way to stop
adding text is to type a
line that contains only
--repeat, only--
and in the first position--
a period.
.
```

The 1,$p subcommand displays the entire contents of the buffer, showing that the **i** subcommand has inserted the new text.

Copying Lines—The t (Transfer) Subcommand

With the **t** (transfer) subcommand, you can copy lines from one place in the buffer and insert the copies elsewhere. The **t** subcommand does not affect the original lines. The general form of the **t** subcommand is *starting line, ending line* t*line to follow.*

To Copy Lines

Enter a subcommand of the form:

*n,m*t*x*

Where:

n is the first line of the group to be copied.
m is the last line of the group to be copied.
x is the line the copied lines are to follow.

To copy lines to the top of the buffer, use 0 as the line number for the copied lines to follow. To copy lines to the bottom of the buffer, use $ as the line number for the copied lines to follow.

In the following example, the `1,3t4` subcommand copies lines 1 through 3, and inserts the copies after line 4:

```
1,3t4
1,$p
The only way to stop
adding text is to type a
line that contains only
--repeat, only--
The only way to stop
adding text is to type a
line that contains only
and in the first position--
a period.
_
```

The `1,$p` subcommand displays the entire contents of the buffer, showing that **ed** has made and inserted the copies, and that the original lines are not affected.

Using System Commands from ed

Sometimes you may find it convenient to use a system command without leaving the **ed** program—perhaps to use one of the commands covered in Chapter 3, "Using the File System" on page 3-1. Use the ! character to leave from the **ed** program temporarily.

To Use a System Command from ed

Enter:

!*command name*

In the following example, the `!ls` command temporarily suspends the **ed** program and runs the **ls** (list) system command (a command that lists the files in the current directory):

```
!ls
afile
bfile
cfile
!
-
```

The **ls** command displays the names of the files in the current directory (`afile`, `bfile`, and `cfile`), and then displays another ! character. The **ls** command is finished, and you can continue to use **ed**.

You can use any system command from within the **ed** program. You can even run another **ed** program, edit a file, and then return to the original **ed** program. From the second **ed** program, you can run a third, use a system command, and so forth.

Ending the ed Program

This completes the introduction to the **ed** program. To save your file and end the **ed** program, do the steps in the following box:

Saving a File and Ending ed

1. Enter:

   ```
   w
   ```

2. Enter:

   ```
   q
   ```

For a full discussion of the **w** and **q** subcommands, see "Saving Text—The w (Write) Subcommand" on page A-9 and "Leaving the **ed** Program—The q (Quit) Subcommand" on page A-11 respectively.

For information about other features of **ed**, see **ed** in *AIX Operating System Commands Reference.*

For information about printing the files you create with **ed**, see Chapter 2, "Displaying and Printing Files" on page 2-1.

Figures

Glossary

access. To obtain data from or put data in storage.

access permission. A group of designations that determine who can access a particular AIX file and how the user may access the file.

account. The log in directory and other information that give a user access to the system.

activity manager. A collection of system-supplied tasks allowing users to manage their activities. Provides the ability to list current activities (Activity List) and to begin, cancel, hide, and activate activities.

All Points Addressable (APA) display. A display that allows each pel to be individually addressed. An APA display allows for images to be displayed that are not made up of images predefined in character boxes. Contrast with *character display.*

allocate. To assign a resource, such as a disk file or a diskette file, to perform a specific task.

alphabetic. Pertaining to a set of letters a through z.

alphanumeric character. Consisting of letters, numbers and often other symbols, such as punctuation marks and mathematical symbols.

American National Standard Code for Information Interchange (ASCII). The code developed by ANSI for information interchange among data processing systems, data communications systems, and associated equipment. The ASCII character set consists of 7-bit control characters and symbolic characters.

American National Standards Institute. An organization sponsored by the Computer and Business Equipment Manufacturers Association for establishing voluntary industry standards.

application. A program or group of programs that apply to a particular business area, such as the Inventory Control or the Accounts Receivable application.

application program. A program used to perform an application or part of an application.

argument. Numbers, letters, or words that change the way a command works.

ASCII. See *American National Standard Code for Information Interchange.*

attribute. A characteristic. For example, the attribute for a displayed field could be blinking.

auto carrier return. The system function that places carrier returns automatically within the text and on the display. This is accomplished by moving whole words that exceed the line end zone to the next line.

backend. The program that sends output to a particular device. There are two types of backends: friendly and unfriendly.

background process. (1) A process that does not require operator intervention that can be run by the computer while the work station is used to do other work. (2) A mode of program execution in which the shell does not wait for program completion before prompting the user for another command.

backup copy. A copy, usually of a file or group of files, that is kept in case the original file or files are unintentionally changed or destroyed.

backup diskette. A diskette containing information copied from a fixed disk or from another diskette. It is used in case the original information becomes unusable.

bad block. A portion of a disk that can never be used reliably.

base address. The beginning address for resolving symbolic references to locations in storage.

base name. The last element to the right of a full path name. A filename specified without its parent directories.

batch printing. Queueing one or more documents to print as a separate job. The operator can type or revise additional documents at the same time. This is a background process.

batch processing. A processing method in which a program or programs process records with little or no operator action. This is a background process. Contrast with *interactive processing.*

binary. (1) Pertaining to a system of numbers to the base two; the binary digits are 0 and 1. (2) Involving a choice of two conditions, such as on-off or yes-no.

bit. Either of the binary digits 0 or 1 used in computers to store information. See also *byte.*

block. (1) A group of records that is recorded or processed as a unit. Same as *physical record.* (2) In data communications, a group of records that is recorded, processed, or sent as a unit. (3) A block is 512 bytes long. (4) A logical block is 2048 bytes long.

block file. A file listing the usage of blocks on a disk.

block special file. A special file that provides access to an input or output device is capable of supporting a file system. See also *character special file.*

bootstrap. A small program that loads larger programs during system initialization.

branch. In a computer program an instruction that selects one of two or more alternative sets of instructions. A conditional branch occurs only when a specified condition is met.

breakpoint. A place in a computer program, usually specified by an instruction, where execution may be interrupted by external intervention or by a monitor program.

buffer. (1) A temporary storage unit, especially one that accepts information at one rate and delivers it at another rate. (2) An area of storage, temporarily reserved for performing input or output, into which data is read, or from which data is written.

burst pages. On continuous-form paper, pages of output that can be separated at the perforations.

byte. The amount of storage required to represent one character; a byte is 8 bits.

call. (1) To activate a program or procedure at its entry point. Compare with *load.*

callouts. An AIX kernel parameter establishing the maximum number of scheduled activities that can be pending simultaneously.

cancel. To end a task before it is completed.

carrier return. (1) In text data, the action causing line ending formatting to be performed at the current cursor location followed by a line advance of the cursor. Equivalent to the carriage return of a typewriter. (2) A keystroke generally indicating the end of a command line.

case sensitive. Able to distinguish between uppercase and lowercase letters.

character. A letter, digit, or other symbol.

character display. A display that uses a character generator to display predefined character boxes of images (characters) on the screen. This kind of display cannot address the screen any less than one character box at a time. Contrast with *All Points Addressable display.*

character key. A keyboard key that allows the user to enter the character shown on the key. Compare with *function keys.*

character position. On a display, each location that a character or symbol can occupy.

character set. A group of characters used for a specific reason; for example, the set of characters a printer can print or a keyboard can support.

character special file. A special file that provides access to an input or output device. The character interface is used for devices that do not use block I/O. See also *block special file.*

character string. A sequence of consecutive characters.

character variable. The name of a character data item whose value may be assigned or changed while the program is running.

child. (1) Pertaining to a secured resource, either a file or library, that uses the user list of a parent resource. A child resource can have only one parent resource. (2) In the AIX Operating System, child is a *process* spawned by a parent process that shares resources of parent process. Contrast with *parent.*

C language. A general-purpose programming language that is the primary language of the AIX Operating System.

class. Pertaining to the I/O characteristics of a device. AIX devices are classified as block or character.

close. (1) To end an activity and remove that window from the display.

code. (1) Instructions for the computer. (2) To write instructions for the computer; to *program.* (3) A representation of a condition, such as an error code.

code segment. See *segment.*

collating sequence. The sequence in which characters are ordered within the computer for sorting, combining, or comparing.

color display. A display device capable of displaying more than two colors and the shades produced via the two colors, as opposed to a monochrome display.

column. A vertical arrangement of text or numbers.

column headings. Text appearing near the top of columns of data for the purpose of identifying or titling.

command. A request to perform an operation or run a program. When parameters, arguments, flags, or other operands are associated with a command, the resulting character string is a single command.

command interpreter. A program that sends instructions to the kernel; also called an interface.

command line. The area of the screen where commands are displayed as they are typed.

command line editing keys. Keys for editing the command line.

command programming language. Facility that allows programming by the combination of commands rather than by writing statements in a conventional programming language.

compile. (1) To translate a program written in a high-level programming language into a machine language program. (2) The computer actions required to transform a source file into an executable object file.

compress. (1) To move files and libraries together on disk to create one continuous area of unused space. (2) In data communications, to delete a series of duplicate characters in a character string.

concatenate. (1) To link together. (2) To join two character strings.

condition. An expression in a program or procedure that can be evaluated to a value of either true or false when the program or procedure is running.

configuration. The group of machines, devices, and programs that make up a computer system. See also *system customization.*

configuration file. A file that specifies the characteristics of a system or subsystem, for example, the AIX queueing system.

consistent. Pertaining to a file system, without internal discrepancies.

console. (1) The main AIX display station. (2) A device name associated with the main AIX display station.

constant. A data item with a value that does not change. Contrast with *variable.*

context search. A search through a file whose target is a character string.

control block. A storage area used by a program to hold control information.

control commands. Commands that allow conditional or looping logic flow in shell procedures.

control program. Part of the AIX Operating System system that determines the order in which basic functions should be performed.

controlled cancel. The system action that ends the job step being run, and saves any new data already created. The job that is running can continue with the next job step.

copy. The action by which the user makes a whole or partial duplicate of already existing data.

crash. An unexpected interruption of computer service, usually due to a serious hardware or software malfunction.

current directory. The directory that is active, and can be displayed with the **pwd** command.

current line. The line on which the cursor is located.

current working directory. See *current directory.*

cursor. (1) A movable symbol (such as an underline) on a display, used to indicate to the operator where the next typed character will be placed or where the next action will be directed. (2) A marker that indicates the current data access location within a file.

cursor movement keys. The directional keys used to move the cursor.

customize. To describe (to the system) the devices, programs, users, and user defaults for a particular data processing system.

cylinder. All fixed disk or diskette tracks that can be read or written without moving the disk drive or diskette drive read/write mechanism.

daemon. See *daemon process.*

daemon process. A process begun by the root or the root shell that can be stopped only by the root. Daemon processes generally provide services that must be available at all times such as sending data to a printer.

data block. See *block.*

data communications. The transmission of data between computers, or remote devices or both (usually over long distance).

data stream. All information (data and control information) transmitted over a data link.

debug. (1) To detect, locate, and correct mistakes in a program. (2) To find the cause of problems detected in software.

default. A value that is used when no alternative is specified by the operator.

default directory. The directory name supplied by the operating system if none is specified.

default drive. The drive name supplied by the operating system if none is specified.

default value. A value stored in the system that is used when no other value is specified.

delete. To remove. For example, to delete a file.

dependent work station. A work station having little or no standalone capability, that must be connected to a host or server in order to provide any meaningful capability to the user.

device. An electrical or electronic machine that is designed for a specific purpose and that attaches to your computer, for example, a printer, plotter, disk drive, and so forth.

device driver. A program that operates a specific device, such as a printer, disk drive, or display.

device name. A name reserved by the system that refers to a specific device.

diagnostic. Pertaining to the detection and isolation of an error.

diagnostic aid. A tool (procedure, program, reference manual) used to detect and isolate a device or program malfunction or error.

diagnostic routine. A computer program that recognizes, locates, and explains either a fault in equipment or a mistake in a computer program.

digit. Any of the numerals from 0 through 9.

directory. A type of file containing the names and controlling information for other files or other directories.

disable. To make nonfunctional.

discipline. Pertaining to the order in which requests are serviced, for example, first-come-first-served (fcfs) or shortest job next (sjn).

disk I/O. Fixed-disk input and output.

diskette. A thin, flexible magnetic plate that is permanently sealed in a protective cover. It can be used to store information copies from the disk or another diskette.

diskette drive. The mechanism used to read and write information on diskettes.

display device. An output unit that gives a visual representation of data.

display screen. The part of the display device that displays information visually.

display station. A device that includes a keyboard from which an operator can send information to the system and a display screen on which an operator can see the information sent to or received from the computer.

dump. (1) To copy the contents of all or part of storage, usually to an output device. (2) Data that has been dumped.

dump diskette. A diskette that contains a dump or is prepared to receive a dump.

dump formatter. Program for analyzing a dump.

EBCDIC. See *extended binary-coded decimal interchange code.*

EBCDIC character. Any one of the symbols included in the 8-bit EBCDIC set.

edit. To modify the form or format of data.

edit buffer. A temporary storage area used by an editor.

editor. A program used to enter and modify programs, text, and other types of documents and data.

emulation. Imitation; for example, when one computer imitates the characteristics of another computer.

enable. To make functional.

enter. To send information to the computer by pressing the **Enter** key.

entry. A single input operation on a work station.

environment. The settings for shell variables and paths set associated with each process. These variables can be modified later by the user.

error-correct backspace. An editing key that performs editing based on a cursor position; the cursor is moved one position toward the beginning of the line, the character at the new cursor location is deleted, and all characters following the cursor are moved one position toward the beginning of the line (to fill the vacancy left by the deleted element).

escape character. A character that suppresses the special meaning of one or more characters that follow.

exit value. A numeric value that a command returns to indicate whether it completed successfully. Some commands return exit values that give other information, such as whether a file exists. Shell programs can test exit values to control branching and looping.

expression. A representation of a value. For example, variables and constants appearing alone or in combination with operators.

extended binary-coded decimal interchange code (EBCDIC). A set of 256 eight-bit characters.

feature. A programming or hardware option, usually available at an extra cost.

field. (1) An area in a record or panel used to contain a particular category of data. (2) The smallest component of a record that can be referred to by a name.

FIFO. See *first-in-first-out.*

file. A collection of related data that is stored and retrieved by an assigned name.

file name. The name used by a program to identify a file. See also *label.*

filename. In DOS, that portion of the file name that precedes the extension.

file specification (filespec). The name and location of a file. A file specification consists of a drive specifier, a path name, and a file name.

file system. The collection of files and file management structures on a physical or logical mass storage device, such as a diskette or minidisk.

filetab. An AIX kernel parameter establishing the maximum number of files that can be open simultaneously.

filter. A command that reads standard input data, modifies the data, and sends it to standard output.

first-in-first-out (FIFO). A named permanent pipe. A FIFO allows two unrelated processes to exchange information using a pipe connection.

fixed disk. A flat, circular, nonremoveable plate with a magnetizable surface layer on which data can be stored by magnetic recording.

fixed-disk drive. The mechanism used to read and write information on fixed disk.

flag. A modifier that appears on a command line with the command name

that defines the action of the command. Flags in the AIX Operating System almost always are preceded by a dash.

font. A family or assortment of characters of a given size and style.

foreground. A mode of program execution in which the shell waits for the program specified on the command line to complete before returning your prompt.

format. (1) A defined arrangement of such things as characters, fields, and lines, usually used for displays, printouts, or files. (2) The pattern which determines how data is recorded.

formatted diskette. A diskette on which control information for a particular computer system has been written but which may or may not contain any data.

free list. A list of available space on each file system. This is sometimes called the free-block list.

free-block list. See *free list.*

full path name. The name of any directory or file expressed as a string of directories and files beginning with the root directory.

function. A synonym for procedure. The C language treats a function as a data type that contains executable code and returns a single value to the calling function.

function keys. Keys that request actions but do not display or print characters. Included are the keys that normally produce a printed character, but when used with the code key produce a function instead. Compare with *character key.*

generation. For some remote systems, the translation of configuration information into machine language.

Gid. See *group number.*

global. Pertains to information available to more than one program or subroutine.

global action. An action having general applicability, independent of the context established by any task.

global character. The special characters * and ? that can be used in a file specification to match one or more characters. For example, placing a ? in a file specification means any character can be in that position.

global search. The process of having the system look through a document for specific characters, words, or groups of characters.

global variable. A symbol defined in one program module, but used in other independently assembled program modules.

graphic character. A character that can be displayed or printed.

group name. A name that uniquely identifies a group of users to the system.

group number (Gid). A unique number assigned to a group of related users. The group number can often be substituted in commands that take a group name as an argument.

hardware. The equipment, as opposed to the programming, of a computer system.

header. Constant text that is formatted to be in the top margin of one or more pages.

header label. A special set of records on a diskette describing the contents of the diskette.

here document. Data contained within a shell program or procedure (also called *inline input*).

highlight. To emphasize an area on the display by any of several methods, such as brightening the area or reversing the color of characters within the area.

history file. A file containing a log of system actions and operator responses.

hog factor. In system accounting, an analysis of how many times each command was run, how much processor time and memory it used, and how intensive that use was.

home directory. (1) A directory associated with an individual user. (2) The user's current directory on login or after issuing the **cd** command with no argument.

I/O. See *input/output.*

ID. Identification.

IF expressions. Expressions within a procedure, used to test for a condition.

indirect block. A block containing pointers to other blocks. Indirect blocks can be single-indirect, double-indirect, or triple-indirect.

informational message. A message providing information to the operator, that does not require a response.

initial program load (IPL). The process of loading the system programs and preparing the system to run jobs. See *initialize.*

initialize. To set counters, switches, addresses, or contents of storage to zero or other starting values at the beginning of, or at prescribed points in, the operation of a computer routine.

inline input. See *here document.*

i-node. The internal structure for managing files in the system. I-nodes contain all of the information pertaining to the node, type, owner, and location of a file. A table of i-nodes is stored near the beginning of a file system.

i-number. A number specifying a particular i-node on a file system.

inodetab. An AIX kernel parameter that establishes a table in memory for storing copies of i-nodes for all active files.

input. Data to be processed.

input device. Physical devices used to provide data to a computer.

input file. A file opened by a program so that the program can read from that file.

input list. A list of variables to which values are assigned from input data.

input redirection. The specification of an input source other than the standard one.

input-output file. A file opened for input and output use.

input-output device number. A value assigned to a device driver by the guest operating system or to the virtual device by the virtual resource manager. This number uniquely identifies the device regardless of whether it is real or virtual.

input/output (I/O). Pertaining to either input, output, or both between a computer and a device.

interactive processing. A processing method in which each system user action causes response from the program or the system. Contrast with *batch processing.*

interface. A shared boundary between two or more entities. An interface might be a hardware component to link two devices together or it might be a portion of storage or registers accessed by two or more computer programs.

interleave factor. Specification of the ratio between contiguous physical blocks (on a fixed-disk) and logically contiguous blocks (as in a file).

interrupt. (1) To temporarily stop a process. (2) In data communications, to take an action at a receiving station that causes the sending station to end a transmission. (3) A signal sent by an I/O device to the processor when an error has occurred or when assistance is needed to complete I/O. An interrupt usually suspends execution of the currently executing program.

IPL. See *initial program load.*

job. (1) A unit of work to be done by a system. (2) One or more related procedures or programs grouped into a procedure.

job queue. A list, on disk, of jobs waiting to be processed by the system.

justify. To print a document with even right and left margins.

kbuffers. An AIX kernel parameter establishing the number of buffers that can be used by the kernel.

K-byte. See *kilobyte.*

kernel. The memory-resident part of the AIX Operating System containing functions needed immediately and frequently. The kernel supervises the input and output, manages and controls the hardware, and schedules the user processes for execution.

kernel parameters. Variables that specify how the kernel allocates certain system resources.

key pad. A physical grouping of keys on a keyboard (for example, numeric key pad, and cursor key pad).

keyboard. An input device consisting of various keys allowing the user to input data, control cursor and pointer locations, and to control the dialog between the user and the display station

keylock feature. A security feature in which a lock and key can be used to restrict the use of the display station.

keyword. One of the predefined words of a programming language; a reserved word.

keyword argument. One type of variable assignment that can be made on the command line.

kill. An AIX Operating System command that stops a process.

kill character. The character that is used to delete a line of characters entered after the user's prompt.

kilobyte. 1024 bytes.

kprocs. An AIX kernel parameter establishing the maximum number of processes that the kernel can run simultaneously.

label. (1) The name in the disk or diskette volume table of contents that identifies a file. See also *file name.* (2) The field of an instruction that assigns a symbolic name to the location at which the instruction begins, or such a symbolic name.

left margin. The area on a page between the left paper edge and the leftmost character position on the page.

left-adjust. The process of aligning lines of text at the left margin or at a tab setting such that the leftmost character in the line or filed is in the leftmost position. Contrast with *right-adjust.*

library. A collection of functions, calls, subroutines, or other data.

licensed program product (LPP). Software programs that remain the property of the manufacturer, for which customers pay a license fee.

line editor. An editor that modifies the contents of a file one line at a time.

linefeed. An ASCII character that causes an output device to move forward one line.

link. A connection between an i-node and one or more file names associated with it.

literal. A symbol or a quantity in a source program that is itself data, rather than a reference to data.

load. (1) To move data or programs into storage. (2) To place a diskette into a diskette drive, or a magazine into a

diskette magazine drive. (3) To insert paper into a printer.

loader. A program that reads run files into main storage, thus preparing them for execution.

local. Pertaining to a device directly connected to your system without the use of a communications line. Contrast with *remote.*

log. To record; for example, to log all messages on the system printer. A list of this type is called a log, such as an error log.

log in. To begin a session at a display station.

log in shell. The program, or command interpreter, started for a user at log in.

log off. To end a session at a display station.

log out. To end a session at a display station.

logical device. A file for conducting input or output with a physical device.

loop. A sequence of instructions performed repeatedly until an ending condition is reached.

main storage. The part of the processing unit where programs are run.

maintenance system. A special version of the AIX Operating System which is loaded from diskette and used to perform system management tasks.

major device number. A system identification number for each device or type of device.

mapped files. Files on the fixed-disk that are accessed as if they are in memory.

mask. A pattern of characters that controls the keeping, deleting, or testing of portions of another pattern of characters.

matrix. An array arranged in rows and columns.

maxprocs. An AIX kernel parameter establishing the maximum number of processes that can be run simultaneously by a user.

memory. Storage on electronic chips. Examples of memory are random access memory, read only memory, or registers. See *storage.*

menu. A displayed list of items from which an operator can make a selection.

message. (1) A response from the system to inform the operator of a condition which may affect further processing of a current program. (2) Information sent from one user in a multi-user operating system to another.

minidisk. A logical division of a fixed disk.

minor device number. A number used to specify various types of information about a particular device, for example, to distinguish among several printers of the same type.

mode word. An i-node field that describes the type and state of the i-node.

modem. See *modulator-demodulator.*

modulation. Changing the frequency or size of one signal by using the frequency or size of another signal.

modulator-demodulator (modem). A device that converts data from the computer to a signal that can be transmitted on a communications line, and converts the signal received to data for the computer.

module. (1) A discrete programming unit that usually performs a specific task or set of tasks. Modules are subroutines and calling programs that are assembled separately, then linked to make a complete program. (2) See *load module.*

mount. To make a file system accessible.

mountab. An AIX kernel parameter establishing the maximum number of file systems that can be mounted simultaneously.

multiprogramming. The processing of two or more programs at the same time.

multivolume file. A diskette file occupying more than one diskette.

nest. To incorporate a structure or structures of some kind into a structure of the same kind. For example, to nest one loop (the nested loop) within another loop (the nesting loop); to nest one subroutine (the nested subroutine) within another subroutine (the nesting subroutine).

network. A collection of products connected by communication lines for information exchange between locations.

new-line character. A control character that causes the print or display position to move to the first position on the next line.

null. Having no value, containing nothing.

null character (NUL). The character hex 00, used to represent the absence of a printed or displayed character.

numeric. Pertaining to any of the digits 0 through 9.

object code. Machine-executable instruction, usually generated by a compiler from source code written in a higher level language. consists of directly executable machine code. For programs that must be linked, object code consists of relocatable machine code.

octal. A base eight numbering system.

open. (1) To make a file available to a program for processing.

operating system. Software that controls the running of programs; in addition, an operating system may provide

services such as resource allocation, scheduling, input/output control, and data management.

operation. A specific action (such as move, add, multiply, load) that the computer performs when requested.

operator. A symbol representing an operation to be done.

output. The result of processing data.

output devices. Physical devices used by a computer to present data to a user.

output file. A file that is opened by a program so that the program can write to that file.

output redirection. The specification of an output destination other than the standard one.

override. (1) A parameter or value that replaces a previous parameter or value. (2) To replace a parameter or value.

overwrite. To write output into a storage or file space that is already occupied by data.

owner. The user who has the highest level of access authority to a data object or action, as defined by the object or action.

pad. To fill unused positions in a field with dummy data, usually zeros or blanks.

page. A block of instructions, data, or both.

page space minidisk. The area on a fixed disk that temporarily stores instructions or data currently being run. See also *minidisk.*

pagination. The process of adjusting text to fit within margins and/or page boundaries.

paging. The action of transferring instructions, data, or both between real storage and external page storage.

parallel processing. The condition in which multiple tasks are being performed simultaneously within the same activity.

parameter. Information that the user supplies to a panel, command, or function.

parent. Pertaining to a secured resource, either a file or library, whose user list is shared with one or more other files or libraries. Contrast with *child.*

parent directory. The directory one level above the current directory.

partition. See *minidisk.*

password. A string of characters that, when entered along with a user identification, allows an operator to sign on to the system.

password security. A program product option that helps prevent the unauthorized use of a display station, by checking the password entered by each operator at sign-on.

path name. See *full path name* and *relative path name.*

pattern-matching character. Special characters such as * or ? that can be used in search patterns. Some used in a file specification to match one or more characters. For example, placing a ? in a file specification means any character can be in that position. Pattern-matching characters are also called wildcards.

permission code. A three-digit octal code, or a nine-letter alphabetic code, indicating the access permissions. The access permissions are read, write, and execute.

permission field. One of the three-character fields within the permissions column of a directory listing indicating the read, write, and run permissions for the file or directory owner, group, and all others.

phase. One of several stages file system checking and repair performed by the **fsck** command.

physical device. See *device.*

physical file. An indexed file containing data for which one or more alternative indexes have been created.

physical record. (1) A group of records recorded or processed as a unit. Same as *block.* (2) A unit of data moved into or out of the computer.

PID. See *process ID.*

pipe. To direct the data so that the output from one process becomes the input to another process.

pipeline. A direct, one-way connection between two or more processes.

pitch. A unit of width of typewriter type, based on the number of times a letter can be set in a linear inch. For example, 10-pitch type has 10 characters per inch.

platen. The support mechanism for paper on a printer, commonly cylindrical, against which printing mechanisms strike to produce an impression.

pointer. A logical connection between physical blocks.

port. (1) To make the programming changes necessary to allow a program that runs on one type of computer to run on another type of computer. (2) An access point for data input to or data output from a computer system. See *connector.*

position. The location of a character in a series, as in a record, a displayed message, or a computer printout.

positional parameter. A shell facility for assigning values from the command line to variables in a program.

print queue. A file containing a list of the names of files waiting to be printed.

printout. Information from the computer produced by a printer.

priority. The relative ranking of items. For example, a job with high priority in the job queue will be run before one with medium or low priority.

priority number. A number that establishes the relative priority of printer requests.

privileged user. The account with superuser authority.

problem determination. The process of identifying why the system is not working. Often this process identifies programs, equipment, data communications facilities, or user errors as the source of the problem.

problem determination procedure. A prescribed sequence of steps aimed at recovery from, or circumvention of, problem conditions.

procedure. See *shell procedure.*

process. (1) A sequence of actions required to produce a desired result. (2) An entity receiving a portion of the processor's time for executing a program. (3) An activity within the system begun by entering a command, running a shell program, or being started by another process.

process accounting. An analysis of the use each process makes of the processing unit, memory, and I/O resources.

process ID (PID). A unique number assigned to a process that is running.

profile. (1) A file containing customized settings for a system or user (2) Data describing the significant features of a user, program, or device.

program. A file containing a set of instructions conforming to a particular programming language syntax.

prompt. A displayed request for information or operator action.

propagation time. The time necessary for a signal to travel from one point on a communications line to another.

qdaemon. The daemon process that maintains a list of outstanding jobs and sends them to the specified device at the appropriate time.

queue. A line or list formed by items waiting to be processed.

queued message. A message from the system that is added to a list of messages stored in a file for viewing by the user at a later time. This is in contrast to a message that is sent directly to the screen for the user to see immediately.

quit. A key, command, or action that tells the system to return to a previous state or stop a process.

quote. To mask the special meaning of certain characters; to cause them to be taken literally.

random access. An access mode in which records can be read from, written to, or removed from a file in any order.

readonly. Pertaining to file system mounting, a condition that allows data to be read, but not modified.

recovery procedure. (1) An action performed by the operator when an error message appears on the display screen. Usually, this action permits the program to continue or permits the operator to run the next job. (2) The method of returning the system to the point where a major system error occurred and running the recent critical jobs again.

redirect. To divert data from a process to a file or device to which it would not normally go.

reference count. In an i-node, a record of the total number of directory entries that refer to the i-node.

relational expression. A logical statement describing the relationship (such as greater than or equal) of two arithmetic expressions or data items.

relational operator. The reserved words or symbols used to express a relational condition or a relational expression.

relative address. An address specified relative to the address of a symbol. When a program is relocated, the addresses themselves will change, but the specification of relative addresses remains the same.

relative addressing. A means of addressing instructions and data areas by designating their locations relative to some symbol.

relative path name. The name of a directory or file expressed as a sequence of directories followed by a file name, beginning from the current directory.

remote. Pertaining to a system or device that is connected to your system through a communications line. Contrast with *local.*

reserved character. A character or symbol that has a special (non-literal) meaning unless quoted.

reserved word. A word that is defined in a programming language for a special purpose, and that must not appear as a user-declared identifier.

reset. To return a device or circuit to a clear state.

restore. To return to an original value or image. For example, to restore a library from diskette.

right adjust. The process of aligning lines of text at the right margin or tab setting such that the rightmost character in the line or file is in the rightmost position.

right justify. See right align.

right margin. The area on a page between the last text character and the right upper edge.

right-adjust. To place or move an entry in a field so that the rightmost character

of the field is in the rightmost position. Contrast with *left-adjust.*

root. Another name sometimes used for superuser.

root directory. The top level of a tree-structured directory system.

root file system. The basic AIX Operating System file system, which contains operating system files and onto which other file systems can be mounted. The root file system is the file system that contains the files that are run to start the system running.

routine. A set of statements in a program causing the system to perform an operation or a series of related operations.

run. To cause a program, utility, or other machine function to be performed.

run-time environment. A collection of subroutines and shell variables that provide commonly used functions and information for system components.

scratch file. A file, usually used as a work file, that exists until the program that uses it ends.

screen. See *display screen.*

scroll. To move information vertically or horizontally to bring into view information that is outside the display screen boundaries.

sector. (1) An area on a disk track or a diskette track reserved to record information. (2) The smallest amount of information that can be written to or read from a disk or diskette during a single read or write operation.

security. The protection of data, system operations, and devices from accidental or intentional ruin, damage, or exposure.

segment. A contiguous area of virtual storage allocated to a job or system task. A program segment can be run by itself, even if the whole program is not in main storage.

separator. A character used to separate parts of a command or file.

sequential access. An access method in which records are read from, written to, or removed from a file based on the logical order of the records in the file.

session records. In the accounting system, a record of time connected and line usage for connected display stations, produced from log in and log out records.

set flags. Flags that can be put into effect with the shell set command.

shared printer. A printer that is used by more than one work station.

shell. See *shell program.*

shell procedure. A series of commands combined in a file that carry out a particular function when the file is run or when the file is specified as an argument to the sh command. Shell procedures are frequently called shell scripts.

shell program. A program that accepts and interprets commands for the operating system (there is an AIX shell program and a DOS shell program).

shell prompt. The character string on the command line indicating the the system can accept a command (typically the $ character).

shell script. See *shell* procedure.

shell variables. Facilities of the shell program for assigning variable values to constant names.

size field. In an i-node, a field that indicates the size, in bytes, of the file associated with the i-node.

software. Programs.

sort. To rearrange some or all of a group of items based upon the contents or characteristics of those items.

source diskette. The diskette containing data to be copied, compared, restored, or backed up.

source program. A set of instructions written in a programming language, that must be translated to machine language compiled before the program can be run.

special character. A character other than an alphabetic or numeric character. For example; *, +, and % are special characters.

special file. Special files are used in the AIX system to provide an interface to input/output devices. There is at least one special file for each device connected to the computer. Contrast with *directory* and *file.* See also *block special file* and *character special file.*

spool files. Files used in the transmission of data among devices.

standalone shell. A limited version of the shell program used for system maintenance.

standalone work station. A work station that can be used to preform tasks independent of (without being connected to) other resources such as servers or host systems.

standard error. The place where many programs place error messages.

standard input. The primary source of data going into a command. Standard input comes from the keyboard unless redirection or piping is used, in which case standard input can be from a file or the output from another command.

standard output. The primary destination of data coming from a command. Standard output goes to the display unless redirection or piping is used, in which case standard output can be to a file or another command.

stanza. A group of lines in a file that together have a common function. Stanzas are usually separated by blank lines, and each stanza has a name.

statement. An instruction in a program or procedure.

status. (1) The current condition or state of a program or device. For example, the status of a printer. (2) The condition of the hardware or software, usually represented in a status code.

storage. (1) The location of saved information. (2) In contrast to memory, the saving of information on physical devices such as disk or tape. See *memory*.

storage device. A device for storing and/or retrieving data.

string. A linear sequence of entities such as characters or physical elements. Examples of strings are alphabetic string, binary element string, bit string, character string, search string, and symbol string.

su. See *superuser*.

subdirectory. A directory contained within another directory in the file system hierarchy.

subprogram. A program invoked by another program, such as a subshell.

subroutine. (1) A sequenced set of statements that may be used in one or more computer programs and at one or more points in a computer program. (2) A routine that can be part of another routine.

subscript. An integer or variable whose value refers to a particular element in a table or an array.

subshell. An instance of the shell program started from an existing shell program.

substring. A part of a character string.

subsystem. A secondary or subordinate system, usually capable of operating independently of, or synchronously with, a controlling system.

superblock. The most critical part of the file system containing information about every allocation or deallocation of a block in the file system.

superuser (su). The user who can operate without the restrictions designed to prevent data loss or damage to the system (User ID 0).

superuser authority. The unrestricted ability to access and modify any part of the operating system associated with the user who manages the system. The authority obtained when one logs in as **root**.

system. The computer and its associated devices and programs.

system call. A request by an active process for a service by the system kernel.

system customization. A process of specifying the devices, programs, and users for a particular data processing system.

system date. The date assigned by the system user during setup and maintained by the system.

system dump. A copy of memory from all active programs (and their associated data) whenever an error stops the system. Contrast with *task dump.*

system management. The tasks involved in maintaining the system in good working order and modifying the system to meet changing requirements.

system parameters. See *kernel parameters.*

system profile. A file containing the default values used in system operations.

system unit. The part of the system that contains the processing unit, the disk drives, and the diskette drives.

system user. A person who uses a computer system.

target diskette. The diskette to be used to receive data from a source diskette.

task. A basic unit of work to be performed. Examples are a user task, a server task, and a processor task.

task dump. A copy of memory from a program that failed (and its associated data). Contrast with *system dump.*

terminal. An input/output device containing a keyboard and either a display device or a printer. Terminals usually are connected to a computer and allow a person to interact with the computer.

text. A type of data consisting of a set of linguistic characters (for example, alphabet, numbers, and symbols) and formatting controls.

text application. A program defined for the purpose of processing text data (for example, memos, reports, and letters).

text editing program. See *editor* and *text application.*

texttab. A kernel parameter establishing the size of the text table, in memory, that contains one entry each active shared program text segment.

trace. To record data that provides a history of events occurring in the system.

trace table. A storage area into which a record of the performance of computer program instructions is stored.

track. A circular path on the surface of a fixed disk or diskette on which information is magnetically recorded and from which recorded information is read.

trap. An unprogrammed, hardware-initiated jump to a specific address. Occurs as a result of an error or certain other conditions.

tree-structured directories. A method for connecting directories such that each directory is listed in another directory except for the root directory, which is at the top of the tree.

truncate. To shorten a field or statement to a specified length.

typematic key. A key that repeats its function multiple times when held down.

typestyle. Characters of a given size, style and design.

Uid. See *user number.*

update. An improvement for some part of the system.

user. The name associated with an account.

user account. See *account.*

user ID. See *user number.*

user name. A name that uniquely identifies a user to the system.

user number (Uid). (1) A unique number identifying an operator to the system. This string of characters limits the functions and information the operator is allowed to use. The Uid can often be substituted in commands that take a user's name as an argument.

user profile. A file containing a description of user characteristics and defaults (for example, printer assignment, formats, group ID) to be conveyed to the system while the user is signed on.

utility. A service; in programming, a program that performs a common service function.

valid. (1) Allowed. (2) True, in conforming to an appropriate standard or authority.

value. (1) In Usability Services, information selected or typed into a pop-up. (2) A set of characters or a quantity associated with a parameter or name. (3) In programming, the contents of a storage location.

variable. A name used to represent a data item whose value can change while the program is running. Contrast with *constant.*

verify. To confirm the correctness of something.

version. Information in addition to an object's name that identifies different modification levels of the same logical object.

virtual device. A device that appears to the user as a separate entity but is actually a shared portion of a real device. For example, several virtual terminals may exist simultaneously, but only one is active at any given time.

virtual machine. A functional simulation of a computer and its related devices.

virtual machine interface (VMI). A software interface between work stations and the operating system. The VMI shields operating system software from hardware changes and low-level interfaces and provides for concurrent execution of multiple virtual machines.

virtual resource manager (VRM). A set of programs that manage the hardware resources (main storage, disk storage, display stations, and printers) of the system so that these resources can be used independently of each other.

virtual resources. See *virtual resource manager.*

virtual storage. Addressable space that appears to be real storage. From virtual storage, instructions and data are mapped into real storage locations.

virtual terminal. Any of several logical equivalents of a display station available at a single physical display station.

Volume ID (Vol ID). A series of characters recorded on the diskette used to identify the diskette to the user and to the system.

VRM. See *virtual resource manager.*

wildcard. See *pattern-matching characters.*

word. A contiguous series of 32 bits (4 bytes) in storage, addressable as a unit. The address of the first byte of a word is evenly divisible by four.

work file. A file used for temporary storage of data being processed.

work station. A device at which an individual may transmit information to, or receive information from, a computer for the purpose of performing a task, for example, a display station or printer. See *programmable work station* and *dependent work station.*

working directory. See *current directory.*

wrap around. Movement of the point of reference in a file from the end of one line to the beginning of the next, or from one end of a file to the other.

Index

Special Characters

A

B

C

D

E

G

Q

R

S

X

The IBM RT PC Family

Reader's Comment Form

Using the AIX Operating System SX23-0794-0

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

For prompt resolution to questions regarding set up, operation, program support, and new program literature, contact the authorized IBM RT PC dealer in your area.

Comments:

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 997, Building 998
11400 Burnet Rd.
Austin, Texas 78758

Fold and tape

Fold and tape

Cut or Fold Along Line

IBM RT PC Using the AIX Operating System

Book Title

SC23-0794

Order No.

Book Evaluation Form

Your comments can help us produce better books. You may use this form to communicate your comments about this book, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Please take a few minutes to evaluate this book as soon as you become familiar with it. Circle Y (Yes) or N (No) for each question that applies and give us any information that may improve this book.

Y N Is the purpose of this book clear?

Y N Is the table of contents helpful?

Y N Is the index complete?

Y N Are the chapter titles and other headings meaningful?

Y N Is the information organized appropriately?

Y N Is the information accurate?

Y N Is the information complete?

Y N Is only necessary information included?

Y N Does the book refer you to the appropriate places for more information?

Y N Are terms defined clearly?

Y N Are terms used consistently?

Y N Are the abbreviations and acronyms understandable?

Y N Are the examples clear?

Y N Are examples provided where they are needed?

Y N Are the illustrations clear?

Y N Is the format of the book (shape, size, color) effective?

Other Comments

What could we do to make this book or the entire set of books for this system easier to use?

Optional Information

Your name

Company name

Street address

City, State, ZIP

No postage necessary if mailed in the U.S.A.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 997, Building 998
11400 Burnet Rd.
Austin, Texas 78758

Fold and tape

Fold and tape

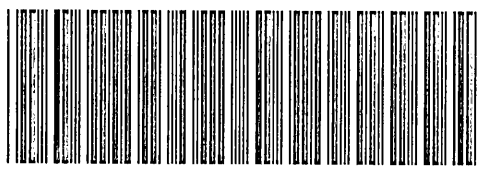
Cut or Fold Along Line

International Business
Machines Corporation
Department 997, Building 998
11400 Burnet Rd.
Austin, Texas 78758

Printed in the
United States of America

SC23-0794-0

SC23-0794-00



92X1266