AIX 5L Version 5.2

IBM

System User’s Guide: Operating System and Devices

AIX 5L Version 5.2

IBM

System User’s Guide: Operating System and Devices

Note

Before using this information and the product it supports, read the information in “Notices” on page 227.

Third Edition (October 2002)

This edition applies to AIX 5L Version 5.2 and to all subsequent releases of this product until otherwise indicated in new editions.

Contents

About This Book

This book contains information for novice system users who want to acquire greater expertise with the operating system. It covers information such as running commands, handling processes, handling files and directories, and printing. In addition, it introduces tasks such as securing files, using storage media, customizing environment files (**.profile**, **.Xdefaults**, **.mwmrc**), and writing shell scripts. For DOS users, this guide presents procedures on using DOS files in this environment.

Users in a networked environment who are interested in learning more about operating system communications commands should read the *AIX 5L Version 5.2 System User's Guide: Communications and Networks*.

Who Should Use This Book

This book is intended for all system users.

Highlighting

The following highlighting conventions are used in this book:

Bold	Identifies commands, keywords, files, directories, and other items whose names are predefined by the system.
Italics	Identifies parameters whose actual names or values are to be supplied by the user.
`Monospace`	Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code similar to what you might write as a programmer, messages from the system, or information you should actually type.

Case-Sensitivity in AIX

Everything in the AIX operating system is case-sensitive, which means that it distinguishes between uppercase and lowercase letters. For example, you can use the **ls** command to list files. If you type `LS`, the system responds that the command is ″not found.″ Likewise, **FILEA**, **FiLea**, and **filea** are three distinct file names, even if they reside in the same directory. To avoid causing undesirable actions to be performed, always ensure that you use the correct case.

ISO 9000

ISO 9000 registered quality systems were used in the development and manufacturing of this product.

Related Publications

The following books contain pertinent information:

- *AIX 5L Version 5.2 System User's Guide: Communications and Networks*
- *AIX 5L Version 5.2 System Management Guide: Operating System and Devices*
- *AIX 5L Version 5.2 System Management Concepts: Operating System and Devices*
- *AIX 5L Version 5.2 Guide to Printers and Printing*
- *AIX 5L Version 5.2 Commands Reference*
- *AIX 5L Version 5.2 Files Reference*

Chapter 1. Login Names, System IDs, and Passwords

The operating system must know who you are in order to provide you with the correct environment. To identify yourself to the operating system, log in by entering your *login name* (also known as your user ID or user name) and a *password*. Passwords are a form of security. People who know your login name cannot log in to your system unless they know your password.

If your system is set up as a multiuser system, each authorized user will have an account, password, and login name on the system. The operating system keeps track of the resources used by each user. This is known as *system accounting*. Each user will be given a private area in the storage space of the system, called the *file system*. When you log in, the file system appears to contain only your files, although there are thousands of other files on the system.

It is possible to have more than one valid login name on a system. If you want to change from one login name to another, you do not have to log out of the system. Rather, you can use the different login names simultaneously in different shells or consecutively in the same shell without logging out. In addition, if your system is part of a network with connections to other systems, you can log in to any of the other systems where you have a login name. This is referred to as a *remote login*.

When you have finished working on the operating system, you log out to ensure that your files and data are secure.

This chapter contains the following sections:

Login and Logout Overview

To use the operating system, your system must be running and you must be logged in. When you log in to the operating system, you identify yourself to the system and allow the system to set up your environment.

This section describes the following procedures:

Logging In to the Operating System

Your system might be set up so that you can only log in during certain hours of the day and on certain days of the week. If you attempt to log in at a time other than the time allowed, your access will be denied. Your system administrator can verify your login times.

You log in at the login prompt. When you log in to the operating system, you are automatically placed into your home directory (also called your *login directory*).

After your system is turned on, log in to the system to start a session.

1. Type your login name following the `login`: prompt and press Enter:

   ```
   login: LoginName
   ```

   For example, if your login name is denise:

   ```
   login: denise
   ```

2. If the `password:` prompt appears, type your password and press Enter. (The screen does not display your password as you type it in.)

   ```
   password: [your password]
   ```

If the password prompt does not appear, you have no password defined; you can begin working in the operating system.

If your machine is not turned on, do the following before you log in:

1. Set the power switches of each attached device to On.
2. Start the system unit by setting the power switch to On (**I**).
3. Look at the three-digit display. When the self-tests complete without error, the three-digit display is blank.

If an error requiring attention occurs, a three-digit code remains, and the system unit stops. See your system administrator for information about error codes and recovery.

When the self-tests complete successfully, a login prompt similar to the following displays on your screen:

```
login:
```

After you have logged in, depending on how your operating system is set up, your system will start up in either a command line interface (shell) or a graphical interface (for example, AIXwindows or Common Desktop Environment (CDE)).

If you have questions concerning the configuration of your password or user name, please consult your system administrator.

Logging in More Than One Time (login Command)

If you are working on more than one project and want to maintain separate accounts, you can have more than one concurrent login.You do this by using the same login name or by using different login names to log in to your system.

Note: Each system has a maximum number of login names that can be active at any given time. This number is determined by your license agreement and varies among installations.

For example, if you are already logged on as denise1 and your other login name is denise2, at the prompt, type:

```
login denise2
```

If the `password:` prompt displays, type your password and press Enter. (The screen does not display your password as you type it.) You now have two logins running on your system.

See the **login** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Becoming Another User on a System (su Command)

You can change the user ID associated with a session (if you know that user's login name)by using the **su** (switch user) command.

For example, if you want to switch and become user joyce, at the prompt, type:

```
su joyce
```

If the `password:` prompt displays, type joyce's password and press Enter. Your user ID is now joyce. If you do not know the password, the request is denied.

To verify that your user ID is joyce, use the **id** command. For more information on the **id** command, see "Displaying User IDs (id Command)" on page 6.

See the **su** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Suppressing Login Messages

After a successful login, the **login** command displays the message of the day, the date and time of the last successful and unsuccessful login attempts for this user, and the total number of unsuccessful login attempts for this user since the last change of authentication information (usually a password). You can suppress these messages by including a **.hushlogin** file in your home directory.

At the prompt in your home directory, type:

```
touch .hushlogin
```

The **touch** command creates the empty file named **.hushlogin** if it does not already exist. The next time you log in, all login messages will be suppressed. You can instruct the system to retain only the message of the day, while suppressing other login messages.

See the **touch** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Logging Out of the Operating System (exit and logout Commands)

To log out of the operating system, do one of the following at the system prompt:

Press the end-of-file control-key sequence (Ctrl-D keys).

OR

Type `exit` and press Enter.

OR

Type `logout` and press Enter.

After you log out, the system displays the `login:` prompt.

Stopping the Operating System (shutdown Command)

> **Attention:** Do not turn off the system without first shutting down. Turning off the system ends all processes running on the system. If other users are working on the system, or if jobs are running in the background, data might be lost. Perform proper shutdown procedures before you stop the system.

If you have root user authority, you can use the **shutdown** command to stop the system. If you are not authorized to use the **shutdown** command, simply log out of the operating system and leave it running.

At the prompt, type:

```
shutdown
```

When the **shutdown** command completes and the operating system stops running, you receive the following message:

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

See the **shutdown** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

User and System Identification

This section describes following procedures available for displaying information that identifies users on your system and the system you are using.

- "Displaying Your Login Name (whoami and logname Commands)"
- "Displaying the Operating System's Name (uname Command)" on page 5
- "Displaying Your System's Name (uname Command)" on page 5
- "Displaying Who Is Logged In (who Command)" on page 6
- "Displaying User IDs (id Command)" on page 6

Displaying Your Login Name (whoami and logname Commands)

When you have more than one concurrent login, it is often easy to lose track of the login names or, in particular, the login name that you are using at the time.

Using the whoami Command

To determine which login name is being used, at the prompt, type:

```
whoami
```

The system displays information similar to the following:

```
denise
```

In this example, the login name being used is `denise`.

See the **whoami** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Using the who am i Command

A variation of the **who** command, the **who am i** command, allows you to display the login name, terminal name, and time of the login.At the prompt, type:

```
who am i
```

The system displays information similar to the following:

```
denise   pts/0   Jun 21 07:53
```

In this example, the login name is `denise`, the name of the terminal is `pts/0`, and this user logged in at 7:53 a.m. on June 21.

See the **who** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Using the logname Command

Another variation of the **who** command, the **logname** command displays the same information as the **who** command.

At the prompt, type:

```
logname
```

The system displays information similar to the following:

```
denise
```

In this example, the login name is `denise`.

Displaying the Operating System's Name (uname Command)

To display the name of the operating system, use the **uname** command .

For example, at the prompt, type:

```
uname
```

The system displays information similar to the following:

```
AIX
```

In this example, the operating system name is AIX.

See the **uname** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Displaying Your System's Name (uname Command)

To display the name of your system if you are on a network, use the **uname** command with the **-n** flag. Your system name identifies your system to the network; it is not the same as your login ID.

For example, at the prompt, type:

```
uname -n
```

The system displays information similar to the following:

```
barnard
```

In this example, the system name is `barnard`.

See the **uname** command in the *AIX 5L Version 5.2 Commands Reference* Book for the complete syntax.

Displaying Who Is Logged In (who Command)

To display information about all users currently on the local system, use the **who** command . The following information is displayed: login name, system name, and date and time of login.

> **Note:** This command only identifies users on the local node.

To display information about who is using the local system node, type:

```
who
```

The system displays information similar to the following:

```
joe   lft/0 Jun 8 08:34
denise   pts/1 Jun 8 07:07
```

In this example, the user `joe`, on terminal `lft/0`, logged in at 8:34 a.m. on June 8.

See the **who** command in the *AIX 5L Version 5.2 Commands Reference* for the exact syntax.

Displaying User IDs (id Command)

To displays the system identifications (IDs) for a specified user, use the **id** command . The system IDs are numbers that identify users and user groups to the system. The **id** command displays the following information, when applicable:

- User name and real user ID
- Name of the user's group and real group ID
- Name of the user's supplementary groups and supplementary group IDs, if any

For example, at the prompt, type:

```
id
```

The system displays information similar to the following:

```
uid=1544(sah) gid=300(build) euid=0(root) egid=9(printq) groups=0(system),10(audit)
```

In this example, the user has user name `sah` with an ID number of `1544`; a primary group name of `build` with an ID number of `300`; an effective user name of `root` with an ID number of `0`; an effective group name of `printq` with an ID number of `9`; and two supplementary group names of `system` and `audit`, with ID numbers `0` and `10`, respectively.

For example, at the prompt, type:

```
id denise
```

The system displays information similar to the following:

```
uid=2988(denise) gid=1(staff)
```

In this example, the user `denise` has an ID number of `2988` and only has a primary group name of `staff` with an ID number of `1`.

See the **id** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Passwords

Your system associates a password with each account. A unique password provides some system security for your files. Security is an important part of computer systems because it keeps unauthorized people from gaining access to the system and from tampering with other users' files. Security can also allow some users exclusive privileges to which commands they can use and which files they can access. For protection, some system administrators permit the users access only to certain commands or files.

This section describes the following procedures:
- "Password Guidelines"
- "Changing Passwords (passwd Command)"
- "Setting Passwords to Null (passwd Command)" on page 8

Password Guidelines

You should have a unique password. *Passwords should not be shared.* Protect passwords as you would any other company asset. When creating passwords, make sure they are difficult to guess, but not so difficult that you have to write them down to remember them.

Using obscure passwords keeps your user ID secure. Passwords based on personal information, such as your name or birthday, are poor passwords. Even common words can be easily guessed.

Good passwords have at least six characters and include nonalphabetic characters. Strange word combinations and words purposely misspelled are also good choices.

Note: If your password is so hard to remember that you have to write it down, it is not a good password.

Use the following guidelines when selecting a password:
- Do not use your user ID as a password. Do not use it reversed, doubled, or otherwise modified.
- Do not reuse passwords. The system might be set up to deny the reuse of passwords.
- Do not use any person's name as your password.
- Do not use words that can be found in the online spelling-check dictionary as your password.
- Do not use passwords shorter than six characters.
- Do not use obscene words; they are some of the first ones checked when guessing passwords.
- Do use passwords that are easy to remember, so you won't have to write them down.
- Do use passwords that use both letters and numbers and that have both lowercase and uppercase letters.
- Do use two words, separated by a number, as a password.
- Do use pronounceable passwords. They are easier to remember.
- Do not write passwords down. However, if you must write them down, place them in a physically secure place, such as a locked cabinet.

Changing Passwords (passwd Command)

To change your password, use the **passwd** command.
1. At the prompt, type:
   ```
   passwd
   ```
   If you do not already have a password, skip step 2.
2. The following prompt displays:

```
Changing password for UserID
UserID's Old password:
```

This request keeps an unauthorized user from changing your password while you are away from your system. Type your current password and press Enter.

3. The following prompt displays:

   ```
   UserID's New password:
   ```

   Type the new password you want and press Enter.

4. The following prompt displays, asking you to reenter your new password.

   ```
   Enter the new password again:
   ```

   This request protects you from setting your password to a mistyped string that you cannot re-create.

See the **passwd** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Setting Passwords to Null (passwd Command)

If you do not want to enter a password each time you log in, set your password to null (blank).

To set your password to null, type:

```
passwd
```

When you are prompted for the new password, press Enter or Ctrl-D.

The **passwd** command does not prompt again for a password entry. A message verifying the null password displays.

See the **passwd** command in the *AIX 5L Version 5.2 Commands Reference* Book for more information and the exact syntax.

Command Summary for Login Names, System IDs, and Passwords

Login and Logout Commands

login	Initiates your session
logout	Stops all your processes
shutdown	Ends system operation
su	Changes the user ID associated with a session
touch	Updates the access and modification times of a file, or creates an empty file

User and System Identification Commands

id	Displays the system identifications of a specified user
logname	Displays login name.
uname	Displays the name of the current operating system
who	Identifies the users currently logged in
whoami	Displays your login name

Password Command

passwd	Changes a user's password

Related Information

For further information on this topic, see the following

- Chapter 4, “Commands and Processes” on page 25
- Chapter 10, “File and System Security” on page 117
- Chapter 2, “User Environment and System Information” on page 11
- Chapter 11, “Customizing the User Environment” on page 129

Related Information

Chapter 4, “Commands and Processes” on page 25

Chapter 10, “File and System Security” on page 117

Chapter 2, “User Environment and System Information” on page 11

Chapter 11, “Customizing the User Environment” on page 129

Chapter 12, “Shells” on page 139

“Korn Shell or POSIX Shell Commands” on page 144

“Bourne Shell” on page 184

“C Shell” on page 200

Chapter 2. User Environment and System Information

Each login name has its own system environment. The system environment is an area where information that is common to all processes running in a session is stored. You can use several commands to display information about your system.

This chapterdiscusses the following procedures for displaying information about your environment.

Listing System Devices (lscfg Command)

To display the name, location, and description of each device found in the current configuration, use the **lscfg** command. The list is sorted by device location.

For example, to list the devices configured in your system, at the prompt, type:

```
lscfg
```

Press Enter.

The system displays output similar to the following:

```
INSTALLED RESOURCE LIST

The following resources are installed on your machine.

+/- = Added/Deleted from Diagnostic Test List.
*   = NOT Supported by Diagnostics.


  Model Architecture: chrp
  Model Implementation: Multiple Processor, PCI bus

+  sysplanar0   00-00           CPU Planar
+  fpa0         00-00           Floating Point Processor
+  mem0         00-0A           Memory Card
+  mem1         00-0B           Memory Card
+  ioplanar0    00-00           I/O Planar
+  rs2320       00-01           RS232 Card
+  tty0         00-01-0-01      RS232 Card Port
-  tty1         00-01-0-02      RS232 Card Port
   ..
   ..
   ..
```

The device list is not sorted by device location alone. It is sorted by the parent/child hierarchy. If the parent has multiple children, the children are sorted by device location. If the children have the same device location, they are displayed in the order in which they were obtained by the software. To display information about a specific device, you can use the **-l** flag. For example, to list the information on device **sysplanar0**, at the prompt, type:

```
lscfg -l sysplanar0
```

Press Enter.

The system displays output similar to the following:

```
DEVICE          LOCATION     DESCRIPTION

sysplanar0      00-00        CPU Planar
```

You can also use the **lscfg** command to display vital product data (VPD), such as part numbers, serial numbers, and engineering change levels. For some devices, the VPD is collected automatically and added to the system configuration. For other devices, the VPD is entered manually. An `ME` preceding the data indicates that the data was entered manually.

For example, to list VPD for devices configured in your system, at the prompt, type:

```
lscfg -v
```

Press Enter.

The system displays output similar to the following:

```
INSTALLED RESOURCE LIST WITH VPD

The following resources are installed in your machine.

  Model Architecture: chrp
  Model Implementation: Multiple Processor, PCI bus

sysplanar0   00-00   CPU Planar

   Part Number.........342522
   EC Level............254921
   Serial Number.......353535

fpa0   00-00   Floating Point Processor
mem0   00-0A   Memory Card

   EC Level............990221
.
.
.
```

See the **lscfg** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Displaying the Console Name (lscons Command)

To write the name of the current console device to standard output (usually your screen), use the **lscons** command.

For example, at the prompt, type:

```
lscons
```

Press Enter.

The system displays output similar to the following:

```
/dev/lft0
```

See the **lscons** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Displaying the Terminal Name (tty Command)

To display the name of your terminal, use the **tty** command.

For example, at the prompt, type:

```
tty
```

Press Enter.

The system displays information similar to the following:

```
/dev/tty06
```

In this example, `tty06` is the name of the terminal, and **/dev/tty06** is the device file that contains the interface to this terminal.

See the **tty** command in the *AIX 5L Version 5.2 Commands Reference* for the exact syntax.

Listing Available Displays (lsdisp Command)

To list the displays currently available on your system, providing a display identification name, slot number, display name, and description of each of the displays, use the **lsdisp** command.

For example, to list all available displays, type:

```
lsdisp
```

Press Enter.

Following is an example of the output. The list displays in ascending order according to slot number.

```
Name   Slot    Name        Description
ppr0   00-01   POWER_G4    Midrange Graphics Adapter
gda0   00-03   colorgda    Color Graphics Display Adapter
ppr1   00-04   POWER_Gt3   Midrange Entry Graphics Adapter
```

See the **lsdisp** command in the *AIX 5L Version 5.2 Commands Reference* for the complet syntax.

Listing Available Fonts (lsfont Command)

To display a list of the fonts available to your display, use the **lsfont** command.

For example, to list all fonts available to the display in list format, type:

```
lsfont
```

Press Enter.

Following is an example of the output, showing the font identifier, file name, glyph size and font encoding:

```
FONT  FILE           GLYPH  FONT
ID    NAME           SIZE   ENCODING
====  =============  =====  =========
0     Erg22.iso1.snf  12x30  ISO8859-1
1     Erg11.iso1.snf   8x15  ISO8859-1
```

See the **lsfont** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Listing the Current Software Keyboard Map (lskbd Command)

To display the absolute path name of the current software keyboard map loaded into the system, use the **lskbd** command.

For example, to list your current keyboard map, type:

```
lskbd
```

Press Enter.

The following is an example of the listing displayed by the **lskbd** command:

```
The current software keyboard map = /usr/lib/nls/loc/C.lftkeymap
```

Listing Available Software Products (lslpp Command)

To display information about software products available for your system, use the **lslpp** command.

For example, to list all the software products in your system, at the system prompt, type:

```
lslpp -l -a
```

Press Enter.

Following is an example of the output:

```
Fileset                Level  State     Description
--------------------  ------- --------  -----------------
Path: /usr/lib/objrepos
   X11_3d.gl.dev.obj           APPLIED   AIXwindows/3D GL
                                        Development Utilities
Fonts
  X11fnt.oldX.fnt             APPLIED   AIXwindows Miscellaneous
                                        X Fonts
X11mEn_US.msg                 APPLIED   AIXwindows NL Message
                                        files

.
.
.
```

If the listing is very long, the top portion may scroll off the screen. To display the listing one page (screen) at a time, use the **lslpp** command piped to the **pg** command. At the prompt, type:

```
lslpp | pg
```

Press Enter.

See the **lslpp** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Listing Control Key Assignments for Your Terminal (stty Command)

To display your terminal settings, use the **stty** command. Note especially which keys your terminal uses for control keys.

For example, at the prompt, type:

```
stty -a
```

Press Enter.

The system displays information similar to the following:

```
.
.
.
intr = ^C; quit = ^\; erase = ^H; kill = ^U; eof = ^D;
eol = ^@ start = ^Q; stop = ^S; susp = ^Z; dsusp = ^Y;
reprint = ^R discard = ^O; werase = ^W; lnext = ^V
.
.
.
```

In this example, lines such as `intr = ^C; quit = ^\; erase = ^H;` are your control key settings. The `^H` key is the Backspace key, and it is set to perform the erase function.

If the listing is very long, the top portion may scroll off the screen. To display the listing one page (screen) at a time, use the **stty** command piped to the **pg** command. At the prompt, type:

```
stty -a | pg
```

Press Enter.

See the **stty** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Listing Environment Variables (env Command)

All variables (with their associated values) known to a command at the beginning of its execution constitute its *environment*. This environment includes variables that a command inherits from its parent process and variables specified as keyword parameters on the command line that calls the command. The shell interacts with the environment in several ways. When started, the shell scans the environment and creates a parameter for each name found, giving the parameter the corresponding value and marking it for export. Executed commands inherit the environment.

To display your current environment variables, use the **env** command. An environment variable that is accessible to all your processes is called a *global variable*.

For example, to list all environment variables, type:

```
env
```

Press Enter.

Following is an example of the output:

```
TMPDIR=/usr/tmp
myid=denise
LANG=En_US
UNAME=barnard
PAGER=/bin/pg
VISUAL=vi
PATH=/usr/ucb:/usr/lpp/X11/bin:/bin:/usr/bin:/etc:/u/denise:/u/denise/bin:/u/bin1
MAILPATH=/usr/mail/denise?denise has mail !!!
MAILRECORD=/u/denise/.Outmail
EXINIT=set beautify noflash nomesg report=1 showmode showmatch
EDITOR=vi
PSCH=>
HISTFILE=/u/denise/.history
LOGNAME=denise
MAIL=/usr/mail/denise
PS1=denise@barnard:${PWD}>
PS3=#
```

```
PS2=>
epath=/usr/bin
USER=denise
SHELL=/bin/ksh
HISTSIZE=500
HOME=/u/denise
FCEDIT=vi
TERM=lft
MAILMSG=**YOU HAVE NEW MAIL. USE THE mail COMMAND TO SEE YOUR PWD=/u/denise
ENV=/u/denise/.env
```

If the listing is very long, the top portion scrolls off the screen. To display the listing one page (screen) at a time, use the **env** command piped to the **pg** command. At the prompt, type:

```
env | pg
```

Press Enter.

See the **env** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Displaying the Value of an Environment Variable (printenv Command)

To display the values of environment variables, use the **printenv** command. If you specify the *Name* parameter, the system only prints the value associated with the parameter you requested. If you do not specify the *Name* parameter, the **printenv** command displays all current environment variables, showing one *Name* **=***Value* sequence per line.

For example, to find the current setting of the **MAILMSG** environment variable, type:

```
printenv MAILMSG
```

Press Enter.

The command returns the value of the **MAILMSG** environment variable. For example:

```
YOU HAVE NEW MAIL
```

See the **printenv** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Working with Bidirectional Languages (aixterm Command)

The **aixterm** command supports Arabic and Hebrew, which are bidirectional languages. Bidirectional languages have the ability to be read and written in two directions, such as from left to right, and from right to left. You can work with Arabic and Hebrew applications by opening a window specifying an Arabic or Hebrew locale.

See the **aixterm** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Command Summary for User Environment and System Information

aixterm	Enables you work with bidirectional languages
env	Displays the current environment or sets the environment for the execution of a command
lscfg	Displays diagnostic information about a device
lscons	Displays the name of the current console
lsdisp	Lists the displays currently available on the system
lsfont	Lists the fonts available for use by the display
lskbd	Lists the keyboard maps currently loaded in the system
lslpp	Lists software products

printenv	Displays the values of environment variables
stty	Displays system settings
tty	Displays the full path name of your terminal

Related Information

- Chapter 4, “Commands and Processes” on page 25
- Chapter 5, “Input and Output Redirection” on page 45
- “User and System Identification” on page 4
- Chapter 11, “Customizing the User Environment” on page 129

Chapter 3. The Common Desktop Environment

With the Common Desktop Environment, you can access networked devices and tools without having to be aware of their location. You can exchange data across applications by simply dragging and dropping objects.

System administrators find many tasks that previously required complex command line syntax can now be done more easily and similarly from platform to platform. They can also maximize their investment in existing hardware and software by configuring centrally and distributing applications to users. They can centrally manage the security, availability, and interoperability of applications for the users they support.

Note: The Common Desktop Environment (CDE) 1.0. Help volumes, web-based documentation, and hardcopy manuals might refer to the desktop as Common Desktop Environment, the AIXwindows desktop, the Common Desktop Environment, CDE 1.0, or simply, the desktop.

Topics covered in this chapter are:

- “Starting and Stopping the Common Desktop Environment”
- “Modifying Desktop Profiles” on page 20
- “Adding and Removing Displays and Terminals for Common Desktop Environment” on page 20
- “Customizing Display Devices for Common Desktop Environment” on page 22

Starting and Stopping the Common Desktop Environment

You can set up the system so that Common Desktop Environment comes up automatically when you start the system, or you can start Common Desktop Environment manually. You must log in as root to perform each of these tasks.

- “Enabling and Disabling Desktop Autostart”
- “Starting Common Desktop Environment Manually”
- “Stopping Common Desktop Environment Manually” on page 20

Enabling and Disabling Desktop Autostart

You may find it more convenient to set up your system to start Common Desktop Environment automatically when the system is turned on. You can do this through the Web-based System Manager (type `wsm`, then select `System`), through the System Management Interface Tool (SMIT), or from a command line.

Prerequisite

You must have root user authority to enable or disable desktop auto-start.

Starting/Stopping the Common Desktop Environment Automatically Tasks

Task	*SMIT Fast Path*	*Command or File*
Enabling the Desktop Auto-Start [1]	**smit dtconfig**	**/usr/dt/bin/dtconfig -e**
Disabling the Desktop Auto-Start [1]	**smit dtconfig**	**/usr/dt/bin/dtconfig -d**

[1]**Note:** Restart the machine after completing this task.

Starting Common Desktop Environment Manually

You can start Common Desktop Environment manually.

Start the Desktop Login Manager Manually

1. Log in to your system as root.
2. At the command line, type:
   ```
   /usr/dt/bin/dtlogin -daemon
   ```

A **Desktop Login** screen is displayed. When you log in, you will start a desktop session.

Stopping Common Desktop Environment Manually

You can stop Common Desktop Environment manually.

Stop the Login Manager Manually

When you manually stop the login manager, all X-servers and desktop sessions that the login manager started are stopped.

1. Open a terminal emulator window and log in as root.
2. Obtain the process ID of the Login Manager by typing the following:
   ```
   cat /var/dt/Xpid
   ```
3. Stop the Login Manager by typing:
   ```
   kill -term process_id
   ```

Modifying Desktop Profiles

When a user logs in to the desktop, the shell environment file (**.profile** or **.login**) is not automatically read. The desktop runs the X-server before the user logs in, so the function provided by the **.profile** file or the **.login** file must be provided by the desktop's login manager.

User-specific environment variables are set in */Home Directory/* **.dtprofile**. A template for this file is located in **/usr/dt/config/sys.dtprofile**. Place variables and shell commands in **.dtprofile** that apply only to the desktop. Add lines to the end of the **.dtprofile** to incorporate the shell environment file.

System-wide environment variables can be set in Login Manager configuration files. For details on configuring environment variables, see the *Common Desktop Environment 1.0: Advanced User's and System Administrator's Guide*.

Adding and Removing Displays and Terminals for Common Desktop Environment

The login manager can be started from a system with a single local bitmap or graphics console. Many other situations are also possible, however (see the following figure). You can start Common Desktop Environment from:

- Local consoles
- Remote consoles
- Bitmap and character-display
- Xterminal systems running on a host system on the network

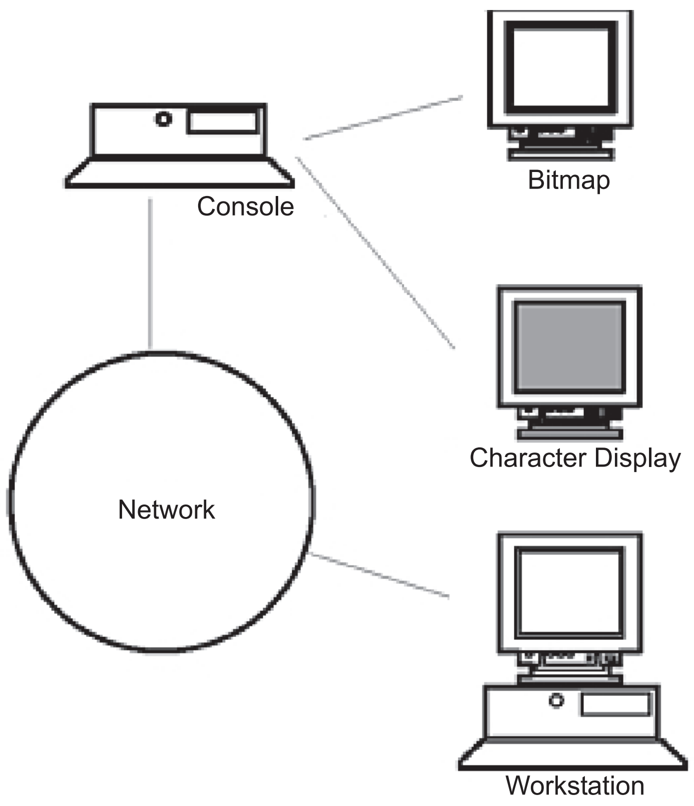

Figure 1. CDE Interface Points. This illustration shows the connection points between a console, a network, a bitmap display, a charactor display, and a workstation.

An Xterminal system consists of a display device, keyboard, and mouse that runs only the Xserver. Clients, including Common Desktop Environment, are run on one or more host systems on the networks. Output from the clients is directed to the Xterminal display.

The following Login Manager configuration tasks support many possible configurations.

- "Removing a Local Display"
- "Adding an ASCII or Character-Display Terminal"

Using a Workstation as an Xterminal

From a command line, type:

```
/usr/bin/X11/X -query hostname
```

The X server of the workstation acting as an Xterminal must:

- Support XDMCP and the **-query** command-line option.
- Provide xhost permission (in **/etc/X*.hosts**) to the terminal host.

Removing a Local Display

To remove a local display, remove its entry in the Xservers file in the **/usr/dt/config** directory.

Adding an ASCII or Character-Display Terminal

A character-display console is a configuration in which the console is not a bitmap device.

Adding an ASCII or Character-Display Console if No Bitmap Display Is Present

1. If the **/etc/dt/config/Xservers** file does not exist, copy the **/usr/dt/config/Xservers** file to the **/etc/dt/config** directory.
2. If you have to copy Xservers to **/etc/dt/config**, you must change or add the **Dtlogin.servers:** line in **/etc/dt/config/Xconfig** to be:

   ```
   Dtlogin*servers: /etc/dt/config/Xservers
   ```
3. Comment out the line in **/etc/dt/config/Xservers** that starts the Xserver. This will disable the Login Option Menu.

   ```
   # * Local local@console /path/X :0
   ```
4. Reread the Login Manager configuration files.

Adding a Character-Display Console if a Bitmap Display Exists

1. If the **/etc/dt/config/Xservers** file does not exist, copy the **/usr/dt/config/Xservers** file to the **/etc/dt/config** directory.
2. If you have to copy Xservers to **/etc/dt/config**, you must change or add the **Dtlogin.servers:** line in **/etc/dt/config/Xconfig** to be:

   ```
   Dtlogin*servers: /etc/dt/config/Xservers
   ```
3. Edit the line in **/etc/dt/config/Xservers** that starts the Xserver to read:

   ```
   * Local local@none /path/X :0
   ```
4. Reread the Login Manager configuration files.

Customizing Display Devices for Common Desktop Environment

You can configure Common Desktop Environment Login Manager to run on systems with two or more display devices.

When a system includes multiple displays, the following configuration requirements must be met:

- A server must be started on each display.
- No Windows mode must be configured for each display.

It might be necessary or desirable to use different dtlogin resources for each display.

It may also be necessary or desirable to use different systemwide environment variables for each display device.

Starting the Server on Each Display Device

1. If the **/etc/dt/config/Xservers** file does not exist, copy the **/usr/dt/config/Xservers** file to the **/etc/dt/config** directory.
2. If you have to copy Xservers to **/etc/dt/config**, you must change the **Dtlogin.servers:** line in **/etc/dt/config/Xconfig** to:

   ```
   Dtlogin*servers: /etc/dt/config/Xservers
   ```
3. Edit **/etc/dt/config/Xservers** to start an X server on each display device.

Syntax

The general syntax for starting the server is:

```
DisplayName DisplayClass DisplayType [ @ite ] Command
```

Only displays with an associated Internal Terminal Emulator (ITE) can operate in No Windows mode. No Windows mode temporarily disables the desktop for the display and runs a getty process if one is not already started. This allows you to log in and perform tasks not possible under Common Desktop

Environment. When you log out, the desktop is restarted for the display device. If a getty is not already running on a display device, Login Manager starts one when No Windows mode is initiated.

Default configuration

When ite is omitted, display:0 is associated with the ITE (/dev/console).

Specifying a Different Display as ITE

- On the ITE display, set ITE to the character device.
- On all other displays, set ITE to none.

Examples

The following entries in the **Xserver** file start a server on three local displays on `sysaaa:0`. Display `:0` will be the console (ITE).

```
sysaaa:0 Local local /usr/bin/X11/X :0
sysaaa:1 Local local /usr/bin/X11/X :1
sysaaa:2 Local local /usr/bin/X11/X :2
```

On host sysbbb, the bitmap display `:0` is not the ITE; the ITE is associated with device **/dev/ttyi1**. The following entries in the **Xserver** file start servers on the two bitmap displays with No Windows Mode enabled on `:1`.

```
sysaaa:0 Local local@none /usr/bin/X11/X :0
sysaaa:1 Local local@ttyi1 /usr/bin/X11/X :1
```

Specifying the Display Name in Xconfig

You cannot use regular hostname:0 syntax for the display name in **/etc/dt/config/Xconfig**.

- Use underscore in place of the colon.
- In a fully qualified host name, use underscores in place of the periods.

Example

```
Dtlogin.claaa_0.resource: value
Dtlogin.sysaaa_prsm_ld_edu_0.resource: value
```

Using Different Login Manager Resources for Each Display

1. If the **/etc/dt/config/Xconfig** file does not exist, copy the **/usr/dt/config/Xconfig** file to the **/etc/dt/config directory**.
2. Use the resources resource in **/etc/dt/config/Xconfig** to specify a different resource file for each display:

   ```
   Dtlogin.DisplayName.resources: path/file
   ```

   whereas *path* is the pathname of the Xresource files to be used and *file*is the file name of the Xresource files to be used.
3. Create each of the resource files specified in the **Xconfig** file. A language specific Xresources file is installed in **/usr/dt/config/<LANG>**.
4. In each file, place the dtlogin resources for that display.

Example

The following lines in the **Xconfig** file specify different resource files for three displays:

```
Dtlogin.sysaaa_0.resources: /etc/dt/config/Xresources0
Dtlogin.sysaaa_1.resources: /etc/dt/config/Xresources1
Dtlogin.sysaaa_2.resources: /etc/dt/config/Xresources2
```

Running Different Scripts for Each Display

1. If the **/etc/dt/config/Xconfig** file does not exist, copy the **/usr/dt/config/Xconfig** file to the **/etc/dt/config** directory.
2. Use the startup, reset, and setup resources in **/etc/dt/config/Xconfig** to specify different scripts for each display (these files are run instead of **Xstartup**, **Xreset**, and **Xsetup.** file):

   ```
   Dtlogin*DisplayName*startup: /path/file
   Dtlogin*DisplayName*reset: /path/file
   Dtlogin*DisplayName*setup: /path/file
   ```

whereas *path* is the pathname of the file to be used and *file*is the file name of the file to be used. The startup script is run as root after the user has logged in, before the Common Desktop Environment session is started.

The script **/usr/dt/config/Xreset** can be used to reverse the setting made in the **Xstartup** file. The **Xreset** file runs when the user logs out.

Example

The following lines in the **Xconfig** file specify different scripts for two displays.

```
Dtlogin.sysaaa_0*startup:   /etc/dt/config/Xstartup0
Dtlogin.sysaaa_1*startup:    /etc/dt/config/Xstartup1
Dtlogin.sysaaa_0*setup:      /etc/dt/config/Xsetup0
Dtlogin.sysaaa_1*setup:      /etc/dt/config/Xsetup1
Dtlogin.sysaaa_0*reset:      /etc/dt/config/Xreset0
Dtlogin.sysaaa_1*reset:      /etc/dt/config/Xreset1
```

Setting Different Systemwide Environment Variables for Each Display

1. If the **/etc/dt/config/Xconfig** file does not exist, copy the **/usr/dt/config/Xconfig** file to the **/etc/dt/config** directory.
2. Set the environment resource in **/etc/dt/config/Xconfig** separately for each display:

   ```
   Dtlogin*DisplayName*environment: value
   ```

The following points apply to environment variables for each display:

- Separate variable assignments with a space or tab.
- Do not use the environment resource to set TZ and LANG.
- There is no shell processing within the **Xconfig** file.

Example

The following lines in the **Xconfig** file set variables for two displays.

```
Dtlogin*syshere_0*environment:EDITOR=vi SB_DISPLAY_ADDR=0xB00000
Dtlogin*syshere_1*environment: EDITOR=emacs \
         SB_DISPLAY_ADDR=0xB00000
```

Chapter 4. Commands and Processes

A *command* is a request to perform an operation or run a program. You use commands to tell the operating system what task you want it to perform. When commands are entered, they are deciphered by a command interpreter (also known as a *shell*) and that task is processed.

A program or command that is actually running on the computer is referred to as a *process*. The operating system can run many different processes at the same time.

The operating system allows you to manipulate the input and output (I/O) of data to and from your system by using specific I/O commands and symbols. You can control input by specifying the location from which to gather data. For example, you can specify to read input while data is entered on the keyboard (standard input) or to read input from a file. You can control output by specifying where to display or store data. For example, you can specify to write output data to the screen (standard output) or to write it to a file.

This chapter discusses the following:

Commands Overview

Some commands can be entered simply by typing one word. It is also possible to combine commands so that the output from one command becomes the input for another command. This is known as *piping*. For more information on piping, see "Shell Features" on page 140.

Flags further define the actions of commands. A *flag* is a modifier used with the command name on the command line, usually preceded by a dash.

Commands can also be grouped together and stored in a file. These are known as *shell procedures* or *shell scripts*. Instead of executing the commands individually, you execute the file that contains the commands. For more information on scripts and procedures, see "Creating and Running a Shell Script" on page 143.

To enter a command, type the command name at the prompt, and press Enter.

```
$ CommandName
```

This section describes the following procedures:

- "Command Syntax"
- "Reading Usage Statements" on page 28
- "Using Web-based System Manager" on page 28
- "Using the smit Command" on page 29
- "Locating a Command or Program (whereis Command)" on page 29
- "Displaying Information about a Command (man Command)" on page 29
- "Displaying the Function of a Command (whatis Command)" on page 30
- "Listing Previously Entered Commands (history Shell Command)" on page 30
- "Repeating Commands Using the history Shell Command" on page 31
- "Substituting Strings Using the history Shell Command" on page 32
- "Editing the Command History" on page 32
- "Creating a Command Alias (alias Shell Command)" on page 33
- "Working with Text-Formatting Commands" on page 34

Command Syntax

Although some commands can be entered by simply typing one word, other commands use flags and parameters. Each command has a syntax that designates both the required and optional flags and parameters. The general format for a command is as follows:

```
CommandName flag(s) parameter(s)
```

The following are some general rules about commands:

- Spaces between commands, flags, and parameters are significant.
- Two commands can be entered on the same line by separating the commands with a semicolon (;). For example:

  ```
  $ CommandOne;CommandTwo
  ```

  The shell runs the commands sequentially.
- Commands are case-sensitive. The shell distinguishes between uppercase and lowercase letters. To the shell, `print` is not the same as `PRINT` or `Print`.

- A very long command can be entered on more than one line by using the backslash (\) character. A backslash signifies line continuation to the shell. The following example is one command that spans two lines:

  ```
  $ ls Mail info temp \
  (press Enter)

  > diary
  (the > prompt appears)
  ```

  The > character is your secondary prompt ($ is the non-root user's default primary prompt), indicating that the current line is the continuation of the previous line. Note that **csh** (the C shell) gives no secondary prompt, and the break must be at a word boundary, and its primary prompt is **%**.

Command Name

The first word of every command is the command name. Some commands have only a command name.

Command Flags

A number of flags might follow the command name. Flags modify the operation of a command and are sometimes called *options*. A flag is set off by spaces or tabs and usually starts with a dash (-). Exceptions are **ps**, **tar**, and **ar**, which do not require a dash in front of some of the flags. For example, in the following command:

```
ls -a -F
```

`ls` is the command name and `-a -F` are the flags.

When a command uses flags, they come directly after the command name. Single-character flags in a command can be combined with one dash. For example, the previous command can also be written as follows:

```
ls -aF
```

There are some circumstances when a parameter actually begins with a dash (-). In this case, use the delimiter dash dash (–) before the parameter. The – tells the command that whatever follows is not a flag but a parameter.

For example, if you wanted to create a directory named `-tmp` and you typed the following command:

```
mkdir -tmp
```

The system displays an error message similar to the following:

```
mkdir: Not a recognized flag: t
Usage: mkdir [-p] [-m mode] Directory ...
```

The correct way of entering the command is as follows:

```
mkdir -- -tmp
```

Your new directory, `-tmp`, is now created.

Command Parameters

After the command name, there might be a number of flags, followed by parameters. Parameters are sometimes called *arguments* or *operands.* Parameters specify information that the command needs in order to run. If you do not specify a parameter, the command might assume a default value. For example, in the following command:

```
ls -a temp
```

`ls` is the command name, `-a` is the flag, and `temp` is the parameter. This command displays all (`-a`) the files in the directory `temp`. In the following example:

```
ls -a
```

the default value is the current directory because no parameter is given. In the following example:

```
ls temp mail
```

no flags are given, and `temp` and `mail` are parameters. In this case, `temp` and `mail` are two different directory names. The **ls** command displays all but the hidden files in each of these directories.

Whenever a parameter or option-argument is, or contains, a numeric value, the number is interpreted as a decimal integer, unless otherwise specified. Numerals in the range 0 to **INT_MAX**, as defined in the **/usr/include/sys/limits.h**file, are syntactically recognized as numeric values.

If a command you want to use accepts negative numbers as parameters or option-arguments, you can use numerals in the range **INT_MIN** to **INT_MAX**, both as defined in the **/usr/include/sys/limits.h** file. This does not necessarily mean that all numbers within that range are semantically correct. Some commands have a built-in specification permitting a smaller range of numbers, for example, some of the print commands. If an error is generated, the error message lets you know the value is out of the supported range, not that the command is syntactically incorrect.

Reading Usage Statements

Usage statements are a way to represent command syntax and consist of symbols such as brackets ([ ]), braces ({ }), and vertical bars (|). The following is a sample of a usage statement for the **unget** command:

unget [**-r***SID*] [**-s**] [**-n**] *File* ...

The following conventions are used in the command usage statements:

- Items that must be entered literally on the command line are in **bold**. These items include the command name, flags, and literal charactors.
- Items representing variables that must be replaced by a name are in *italics*. These items include parameters that follow flags and parameters that the command reads, such as *Files* and *Directories*.
- Parameters enclosed in brackets are optional.
- Parameters enclosed in braces are required.
- Parameters not enclosed in either brackets or braces are required.
- A vertical bar signifies that you choose only one parameter. For example, `[ a | b ]` indicates that you *can* choose a, b, or nothing. Similarly, `{ a | b }` indicates that you *must* choose either a or b.
- Ellipses (`...`) signify the parameter can be repeated on the command line.
- The dash (`-`) represents standard input.

Using Web-based System Manager

Web-based System Manager is a graphical user interface for managing the system, either from a locally attached display or remotely from another system or personal computer equipped with a Web browser. You can start Web-based System Manager in a variety of ways:

- From a command line terminal in the Common Desktop Environment (CDE) by entering the **wsm** command.
- From a command line terminal in the AIXwindows environment by entering the **wsm** command.
- From the CDE Application Manager by going to the System_Admin folder and clicking the **Management Console** icon.
- From an HTML 3.2-compatible Web browser on a personal computer that is configured as described in the *AIX 5L Version 5.2 Web-based System Manager Administration Guide*.

Using the smit Command

The **smit** command is a tool you can use to run other commands. Command names entered as a parameter to the **smit** command might take you to a submenu or panel for that command. For example, **smit lsuser** command takes you directly to **List All Users**, which lists the attributes of users on your system.

See the **smit** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Locating a Command or Program (whereis Command)

The **whereis** command locates the source, binary, and manuals sections for specified files. The command attempts to find the desired program from a list of standard locations.

To find files in the current directory that have no documentation, type:

```
whereis -m -u *
```

Press Enter.

To find all of the files that contain the name `Mail`, type:

```
whereis Mail
```

Press Enter.

The system displays information similar to the following:

```
Mail: /usr/bin/Mail /usr/lib/Mail.rc
```

See the **whereis** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Displaying Information about a Command (man Command)

The **man** command displays information on commands, subroutines, and files. The general format for the **man** command is as follows:

```
man CommandName
```

To obtain information about the **pg** command, type:

```
man pg
```

Press Enter.

The system displays information similar to the following:

```
  pg Command

  Purpose

  Formats files to the display.

  Syntax

  pg [ - Number ] [ -c ] [ -e ] [ -f ] [ -n ] [ -p String ]
  [ -s ] [ +LineNumber | +/Pattern/ ] [ File ... ]

  Description

  The pg command reads a file name from the File parameter and
  writes the file to standard output one screen at a time. If you
  specify a - (dash) as the File parameter, or run the pg command
  without options, the pg command reads standard input. Each
  screen is followed by a prompt. If you press the Enter key,
  another page is displayed. Subcommands used with the pg command
  let you review or search in the file.
```

See the **man** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Displaying the Function of a Command (whatis Command)

The **whatis** command looks up a given command, system call, library function, or special file name, as specified by the *Command* parameter, from a database you create using the **catman -w** command. The **whatis** command displays the header line from the manual section. You can then issue the **man** command to obtain additional information.

The **whatis** command is equivalent to using the **man -f** command.

To find out what the **ls** command does, type:

```
whatis ls
```

Press Enter.

The system displays information similar to the following:

```
ls(1)  -Displays the contents of a directory.
```

See the **whatis** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Listing Previously Entered Commands (history Shell Command)

The **history** command is a Korn shell built-in that lists the last 16 commands entered. The Korn shell saves commands that you entered to a command history file, usually named **$HOME/.sh_history**. This action saves time when you need to repeat a previous command.

By default, the Korn shell saves the text of the last 128 commands. The history file size (specified by the **HISTSIZE** environment variable) is not limited, although a very large history file size can cause the Korn shell to start slowly.

Note: The Bourne shell does not support command history.

For detailed information about shells, see Chapter 12, "Shells" on page 139.

To list the previous commands you entered, at the prompt, type:

```
history
```

Press Enter.

The **history** command entered by itself lists the previous 16 commands entered. The system displays information similar to the following:

```
928   ls
929   mail
930   printenv MAILMSG
931   whereis Mail
932   whatis ls
933   cd /usr/include/sys
934   ls
935   man pg
936   cd
```

```
937   ls | pg
938   lscons
939   tty
940   ls *.txt
941   printenv MAILMSG
942   pwd
943   history
```

The listing first displays the position of the command in the **$HOME/.sh_history** file followed by the command.

To list the previous five commands, at the prompt, type:

```
history -5
```

Press Enter.

A listing similar to the following displays:

```
939   tty
940   ls *.txt
941   printenv MAILMSG
942   pwd
943   history
944   history -5
```

The **history** command followed by a number lists all the previous commands entered, starting at that number.

To list the commands since 938, at the prompt, type:

```
history 938
```

Press Enter.

A listing similar to the following displays:

```
938   lscons
939   tty
940   ls *.txt
941   printenv MAILMSG
942   pwd
943   history
944   history -5
945   history 938
```

Repeating Commands Using the history Shell Command

Use the **r** Korn shell alias to repeat previous commands. Type **r** and press Enter, and you can specify the number or the first character or characters of the command.

If you want to list the displays currently available on the system, type **lsdisp** and press Enter at the prompt. The system returns the information on the screen. If you want the same information returned to you again, at the prompt, type:

```
r
```

Press Enter.

The system runs the most recently entered command again. In this example, the **lsdisp** command runs.

To repeat the **ls *.txt** command, at the prompt, type:

```
r ls
```

Press Enter.

The **r** Korn shell alias locates the most recent command that begins with the character or characters specified.

Substituting Strings Using the history Shell Command

You can also use the **r** Korn shell alias to modify a command before it is run. In this case, a substitution parameter of the form *Old*=*New* can be used to modify the command before it is run.

For example, if command line 940 is **ls *.txt**, and you want to run **ls *.exe**, at the prompt, type:

```
r txt=exe 940
```

Press Enter.

This runs command 940, substituting **exe** for **txt**.

For example, if the command on line 940 is the most recent command that starts with a lowercase letter *l*, you can also type:

```
r txt=exe l
```

Press Enter.

> **Note:** Only the first occurrence of the *Old* string is replaced by the *New* string. Entering the **r** Korn shell alias without a specific command number or character does the substitution to the previous command entered.

Editing the Command History

Use the **fc** Korn shell built-in command to list or edit portions of the command history file. To select a portion of the file to edit or list, specify the number or the first character or characters of the command. You can specify a single command or range of commands.

If you do not specify an editor program as an argument to the **fc** Korn shell built-in command, the editor specified by the **FCEDIT** variable is used. If the **FCEDIT** variable is not defined, the **/usr/bin/ed** editor is used. The edited command or commands are printed and run when you exit the editor. Use the **printenv** command to display the value of the **FCEDIT** variable.

For example, if you want to run the command:

```
cd /usr/tmp
```

which is very similar to command line 933, at the prompt type:

```
fc 933
```

Press Enter.

At this point, your default editor appears with the command line 933. You would change `include/sys` to `tmp`, and when you exit your editor, the edited command is run.

You can also specify the editor you want to use in the **fc** command.

For example, if you want to edit a command using the `/usr/bin/vi` editor, at the prompt, type:

```
fc -e vi 933
```

Press Enter.

At this point, the **vi** editor appears with the command line 933.

You can also specify a range of commands to edit.

For example, if you want to edit the commands 930 through 940, at the prompt, type:

```
fc 930 940
```

Press Enter.

At this point, your default editor appears with the command lines 930 through 940. When you exit the editor, all the commands that appear in your editor are run sequentially.

Creating a Command Alias (alias Shell Command)

An *alias* lets you create a shortcut name for a command, a file name, or any shell text. By using aliases, you save a lot of time when doing tasks you do frequently. The **alias** Korn shell built-in command defines a word as an alias for some command. You can use aliases to redefine built-in commands but not to redefine reserved words.

The first character of an alias name can be any printable character except the metacharacters. Any remaining characters must be the same as for a valid file name.

The format for creating an alias is as follows:

```
alias Name=String
```

in which the *Name* parameter specifies the name of the alias and the *String* parameter specifies a string of characters. If *String* contains blank spaces, enclose it in quotation marks.

To create an alias for the command **rm -i** (prompts you before deleting files), at the prompt, type:

```
alias rm="/usr/bin/rm -i"
```

Press Enter.

In this example, whenever you type the command **rm** and press Enter, the actual command performed is **/usr/bin/rm -i**.

To create an alias for the command **ls -alF | pg** (displays detailed information of all the files in the current directory, including the invisible files; marks executable files with an * and directories with a /; and scrolls per screen), at the prompt, type:

```
alias dir="/usr/bin/ls -alF | pg"
```

Press Enter.

In this example, whenever you type the command **dir** and press Enter, the actual command performed is **/usr/bin/ls -alF | pg**.

To display all the aliases you have, at the prompt, type:

```
alias
```

Press Enter.

The system displays information similar to the following:

```
rm="/usr/bin/rm -i"
dir="/usr/bin/ls -alF | pg"
```

Working with Text-Formatting Commands

You can use text-formatting commands to work with text composed of the international extended character set used for European languages.

International Character Support in Text Formatting

The international extended character set provides the characters and symbols used in many European languages, as well as an ASCII subset composed of English-language characters, digits, and punctuation.

All characters in the European extended character set have ASCII forms. These forms can be used to represent the extended characters in input, or the characters can be entered directly with a device such as a keyboard that supports the European extended characters.

The following text-formatting commands support all international languages that use single-byte characters. These commands are located in **/usr/bin**. (The commands identified with an asterisk (*) support text processing for multibyte languages. For more information on multibyte languages, see "Multibyte Character Support in Text Formatting" on page 35.)

```
addbib*         hyphen          pic*            pstext
checkmm         ibm3812         ps4014          refer*
checknr*        ibm3816         ps630           roffbib*
col*            ibm5587G*       psbanne         soelim*
colcrt          ibm5585H-T*     psdit           sortbib*
deroff*         indxbib*        psplot          tbl*
enscript        lookbib*        psrev           troff*
eqn*            makedev*        psroff          vgrind
grap*           neqn*           psrv            xpreview*
hplj            nroff*
```

Text-formatting commands and macro packages not in the preceding list have not been enabled to process international characters.

Entering Extended Single-Byte Characters

If your input device supports characters from the European-language extended character set, you can enter them directly. Otherwise, use the following ASCII escape sequence form to represent these characters:

The form \[*N*], where *N* is the 2- or 4-digit hexadecimal code for the character.

Note: The NCesc form \<xx> is no longer supported.

Text containing extended characters is output according to the formatting conventions of the language in use. Characters that are not defined for the interface to a specific output device produce no output or error indication.

Although the names of the requests, macro packages, and commands are based on English, most of them can accept input (such as file names and parameters) containing characters in the European extended character set.

For the **nroff** and **troff** commands and their preprocessors, the command input must be ASCII, or an unrecoverable syntax error will result. International characters, either single-byte or multibyte, can be entered when enclosed within quotation marks and within other text to be formatted. For example, using macros from the **pic** command:

```
define foobar % SomeText %
```

After the `define` directive, the first name, `foobar`, must be ASCII. However, the replacement text, *`SomeText`*, can contain non-ASCII characters.

Multibyte Character Support in Text Formatting

Certain text-formatting commands can be used to process text for multibyte languages. These commands are identified with an asterisk (*) in the list under "International Character Support in Text Formatting" on page 34. Text-formatting commands not in the list have not been enabled to process international characters.

Entering Multibyte Characters

If supported by your input device, multibyte characters can be entered directly. Otherwise, you can enter any multibyte character in the ASCII form \[*N*], where ***N*** is the 2-, 4-, 6-, 7-, or 8-digit hexadecimal encoding for the character.

Although the names of the requests, macros, and commands are based on English, most of them can accept input (such as file names and parameters) containing any type of multibyte character.

If you are already familiar with using text-formatting commands with single-byte text, the following list summarizes characteristics that are noteworthy or unique to the multibyte locales:
- Text is not hyphenated.
- Special format types are required for multibyte numerical output. Japanese format types are available.
- Text is output in horizontal lines, filled from left to right.
- Character spacing is constant, so characters automatically align in columns.
- Characters that are not defined for the interface to a specific output device produce no output or error indication.

Processes Overview

A program or command that is actually running on the computer is referred to as a *process*. Processes exist in parent-child hierarchies. A process started by a program or command is a *parent process*; a *child process* is the product of the parent process. A parent process can have several child processes, but a child process can have only one parent.

The system assigns a process identification number (PID number) to each process when it starts. If you start the same program several times, it will have a different PID number each time.

When a process is started on a system, the process uses a part of the available system resources. When more than one process is running, a scheduler that is built into the operating system gives each process its share of the computer's time, based on established priorities. These priorities can be changed by using the **nice** or **renice** commands.

> **Note:** To change a process priority to a higher one, you must have root user authority. All users can lower priorities on a process they start by using the **nice** command, or on a process they have already started, by using the **renice** command.

This section describes the following procedures:
- "Foreground and Background Processes" on page 36
- "Daemons" on page 36
- "Zombie Process" on page 36
- "Starting a Process" on page 36
- "Checking Processes (ps Command)" on page 37
- "Setting the Initial Priority of a Process (nice Command)" on page 38

- "Changing the Priority of a Running Process (renice Command)" on page 39
- "Canceling a Foreground Process" on page 39
- "Stopping a Foreground Process" on page 40
- "Restarting a Stopped Process" on page 40
- "Scheduling a Process for Later Operation (at Command)" on page 40
- "Listing All Scheduled Processes (at or atq Command)" on page 41
- "Removing a Process from the Schedule (at Command)" on page 42
- "Removing a Background Process (kill Command)" on page 42

Foreground and Background Processes

Processes that require a user to start them or to interact with them are called *foreground processes*. Processes that are run independently of a user are referred to as *background processes*. Programs and commands run as foreground processes by default. To run a process in the background, place an ampersand (&) at the end of the command name that you use to start the process.

Daemons

Daemons are processes that run unattended. They are constantly in the background and are available at all times. Daemons are usually started when the system starts, and they run until the system stops. A daemon process performs system services and is available at all times to more than one task or user. Daemon processes are started by the root user or root shell and can be stopped only by the root user. For example, the **qdaemon** process provides access to system resources such as printers. Another common daemon is the **sendmail** daemon.

Zombie Process

A *zombie process* is a dead process that is no longer executing but is still recognized in the process table (in other words, it has a PID number). It has no other system space allocated to it. Zombie processes have been killed or have exited and continue to exist in the process table until the parent process dies or the system is shut down and restarted. Zombie processes display as `<defunct>` when listed by the **ps** command.

Starting a Process

You start a foreground process from a display station by either entering a program name or command name at the system prompt. After a foreground process has started, the process interacts with you at your display station until it is complete. This means no other interaction (for example, entering another command) can take place at the display station until the process is finished or you halt it.

A single user can run more than one process at a time, up to a default maximum of 40 processes per user.

To Start a Process in the Foreground

To run a process in the foreground, type the name of the command with all the appropriate parameters and flags:

```
$ CommandName
```

Press Enter.

To Start a Process in the Background

To run a process in the background, type the name of the command with all the appropriate parameters and flags, followed by an ampersand (&):

```
$ CommandName&
```

Press Enter.

When the process is running in the background, you can perform additional tasks by entering other commands at your display station.

Generally, background processes are most useful for commands that take a long time to run. However, because they increase the total amount of work the processor is doing, background processes also slow down the rest of the system.

Most processes direct their output to standard output, even when they run in the background. Unless redirected, standard output goes to the display device. 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. You can then look at the output whenever you are ready.

> **Note:** Under certain circumstances, a process might generate its output in a different sequence when run in the background than when run in the foreground. Programmers might want to use the **fflush** subroutine to ensure that output occurs in the correct order regardless of whether the process runs in foreground or background.

As long as a background process is running, you can check its status with the **ps** command.

Checking Processes (ps Command)

Any time the system is running, several processes are also running. You can use the **ps** command to find out which processes are running and to display information about those processes.

ps Command

The **ps** command has several flags that enable you to specify which processes to list and what information to display about each process.

To show all processes running on your system, at the prompt, type:

```
ps -ef
```

Press Enter.

The system displays information similar to the following:

```
USER   PID  PPID   C    STIME    TTY  TIME CMD
root     1     0   0   Jun 28      -  3:23 /etc/init
root  1588  6963   0   Jun 28      -  0:02 /usr/etc/biod 6
root  2280     1   0   Jun 28      -  1:39 /etc/syncd 60
mary  2413 16998   2 07:57:30      -  0:05 aixterm
mary 11632 16998   0 07:57:31  lft/1  0:01 xbiff
mary 16260  2413   1 07:57:35  pts/1  0:00 /bin/ksh
mary 16469     1   0 07:57:12  lft/1  0:00 ksh /usr/lpp/X11/bin/xinit
mary 19402 16260  20 09:37:21  pts/1  0:00 ps -ef
```

The columns in the previous output are defined as follows:

USER	User login name
PID	Process ID
PPID	Parent process ID
C	CPU utilization of process
STIME	Start time of process
TTY	Controlling workstation for the process
TIME	Total execution time for the process

CMD Command

In the previous example, the process ID for the **ps -ef**command is 19402. Its parent process ID is 16260, the **/bin/ksh** command.

If the listing is very long, the top portion scrolls off the screen. To display the listing one page (screen) at a time, use the **ps** command piped to the **pg** command. At the prompt, type:

```
ps -ef | pg
```

Press Enter.

To show status information of all processes running on your system, at the prompt, type:

```
ps gv
```

Press Enter.

This form of the command lists a number of statistics for each active process. Output from this command looks similar to the following:

```
   PID    TTY STAT  TIME PGIN  SIZE   RSS   LIM  TSIZ   TRS %CPU %MEM COMMAND
     0      - A     0:44    7     8     8    xx     0     0  0.0  0.0 swapper
     1      - A     1:29  518   244   140    xx    21    24  0.1  1.0 /etc/init
   771      - A     1:22    0    16    16    xx     0     0  0.0  0.0 kproc
  1028      - A     0:00   10    16     8    xx     0     0  0.0  0.0 kproc
  1503      - A     0:33  127    16     8    xx     0     0  0.0  0.0 kproc
  1679      - A     1:03  282   192    12 32768   130     0  0.7  0.0 pcidossvr
  2089      - A     0:22  918    72    28    xx     1     4  0.0  0.0 /etc/sync
  2784      - A     0:00    9    16     8    xx     0     0  0.0  0.0 kproc
  2816      - A     5:59 6436  2664   616     8   852   156  0.4  4.0 /usr/lpp/
  3115      - A     0:27  955   264   128    xx    39    36  0.0  1.0 /usr/lib/
  3451      - A     0:00    0    16     8    xx     0     0  0.0  0.0 kproc
  3812      - A     0:00   21   128    12 32768    34     0  0.0  0.0 usr/lib/lpd/
  3970      - A     0:00    0    16     8    xx     0     0  0.0  0.0 kproc
  4267      - A     0:01  169   132    72 32768    16    16  0.0  0.0 /etc/sysl
  4514  lft/0 A     0:00   60   200    72    xx    39    60  0.0  0.0 /etc/gett
  4776  pts/3 A     0:02  250   108   280     8   303   268  0.0  2.0 -ksh
  5050      - A     0:09 1200   424   132 32768   243    56  0.0  1.0 /usr/sbin
  5322      - A     0:27 1299   156   192    xx    24    24  0.0  1.0 /etc/cron
  5590      - A     0:00    2   100    12 32768    11     0  0.0  0.0 /etc/writ
  5749      - A     0:00    0   208    12    xx    13     0  0.0  0.0 /usr/lpp/
  6111      - T     0:00   66   108    12 32768    47     0  0.0  0.0 /usr/lpp/
```

See the **ps** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Setting the Initial Priority of a Process (nice Command)

You can set the initial priority of a process to a value lower than the base scheduling priority by using the **nice** command to start the process.

> **Note:** To run a process at a higher priority, you must have root user authority.

nice Command

To set the initial priority of a process, type:

```
nice -n Number CommandString
```

where *Number* is in the range of 0 to 39, with 39 being the lowest priority. The *nice value* is the decimal value of the system-scheduling priority of a process. The higher the number, the lower the priority. If you use zero, the process will run at its base scheduling priority. *CommandString* is the command and flags and parameters you want to run.

See the **nice** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

You can also use the **smit nice** command to perform this task.

Changing the Priority of a Running Process (renice Command)

You can change the scheduling priority of a running process to a value lower or higher than the base scheduling priority by using the **renice** command from the command line. This command changes the nice value of a process.

> **Note:** To run a process at a higher priority or to change the priority for a process that you did not start, you must have root user authority.

From the Command Line

To change the initial priority of a running process, type:

```
renice Priority -p ProcessID
```

where *Priority* is in the range of -20 to 20. The higher the number, the lower the priority. If you use zero, the process will run at its base scheduling priority. *ProcessID* is the PID for which you want to change the priority.

You can also use the **smit renice** command to perform this task.

Canceling a Foreground Process

If you start a foreground process and then decide that you do not want it to finish, you can cancel it by pressing INTERRUPT. This is usually Ctrl-C or Ctrl-Backspace. To find out what your INTERRUPT key is set to, see “Listing Control Key Assignments for Your Terminal (stty Command)” on page 14.

> **Note:** INTERRUPT (Ctrl-C) does not cancel background processes. To cancel a background process, you must use the **kill** command.

Most simple 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. The examples in this section, therefore, use a command that takes more than a few seconds to run: **find / -type f**. This command displays the path names for all files on your system. You do not need to study the **find** command in order to complete this section; it is used here simply to demonstrate how to work with processes.

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

```
$ find / -type f
/usr/sbin/acct/lastlogin
/usr/sbin/acct/prctmp
/usr/sbin/acct/prdaily
/usr/sbin/acct/runacct
/usr/sbin/acct/sdisk
/usr/sbin/acct/shutacct INTERRUPT (Ctrl-C)
$ _
```

The system returns the prompt to the screen. Now you can enter another command.

Stopping a Foreground Process

It is possible for a process to be stopped but not have its process ID (PID) removed from the process table. You can stop a foreground process by pressing Ctrl-Z from the keyboard.

> **Note:** Ctrl-Z works successfully in the Korn shell (**ksh**) and C shell (**csh**), but not in the Bourne shell (**bsh**).

Restarting a Stopped Process

This procedure describes how to restart a process that has been stopped with a Ctrl-Z.

> **Note:** Ctrl-Z works successfully in the Korn shell (**ksh**) and C shell (**csh**), but not in the Bourne shell (**bsh**). To restart a stopped process, you must either be the user who started the process or have root user authority.

1. To show all the processes running or stopped but not those killed on your system, type:

   ```
   ps -ef
   ```

   You might want to pipe this command through a **grep** command to restrict the list to those processes most likely to be the one you want to restart. For example, if you want to restart a **vi** session, you could type:

   ```
   ps -ef | grep vi
   ```

   Press Enter. This command would display only those lines from the **ps** command output that contained the word `vi`. The output would look something like this:

   ```
   UID    PID   PPID   C      STIME       TTY  TIME  COMMAND
   root   1234  13682  0      00:59:53    -    0:01  vi test
   root  14277  13682  1      01:00:34    -    0:00  grep vi
   ```
2. In the **ps** command output, find the process you want to restart and note its PID number. In the example, the PID is `1234`.
3. To send the CONTINUE signal to the stopped process, type:

   ```
   kill -19 1234
   ```

   Substitute the PID of your process for the `1234`. The `-19` indicates the CONTINUE signal. This command restarts the process in the background. If the process can run in the background, you are finished with the procedure. If the process must run in the foreground (as a **vi** session would), you must proceed with the next step.
4. To bring the process in to the foreground, type:

   ```
   fg 1234
   ```

   Once again, substitute the PID of your process for the `1234`. Your process should now be running in the foreground. (You are now in your **vi** edit session).

Scheduling a Process for Later Operation (at Command)

You can set up a process as a *batch process* to run in the background at a scheduled time. The **at** and **smit** commands let you enter the names of commands to be run at a later time and allow you to specify when the commands should be run.

> **Note:** The **/var/adm/cron/at.allow** and **/var/adm/cron/at.deny** files control whether you can use the **at** command. A person with root user authority can create, edit, or delete these files. Entries in these files are user login names with one name to a line. The following is an example of an **at.allow** file:

```
root
nick
dee
sarah
```

If the **at.allow** file exists, only users whose login names are listed in it can use the **at** command. A system administrator can explicitly stop a user from using the **at** command by listing the user's login name, in the **at.deny** file. If only the **at.deny** file exists, any user whose name does not appear in the file can use the **at** command.

You cannot use the **at** command if any one of the following is true:
- The **at.allow** file and the **at.deny** file do not exist (allows root user only).
- The **at.allow** file exists but the user's login name is not listed in it.
- The **at.deny** file exists and the user's login name is listed in it.

If the **at.allow** file does not exist and the **at.deny** file does not exist or is empty, only someone with root user authority can submit a job with the **at** command.

The **at** command syntax allows you to specify a date string, a time and day string, or an increment string for when you want the process to run. It also allows you to specify which shell or queue to use. The following examples show some typical uses of the command.

at Command

For example, if your login name is joyce and you have a script named WorkReport that you want to run at midnight, do the following:
1. Type the time you want the program to start running.
   ```
   at midnight
   ```
2. Type the names of the programs to run, pressing Enter after each name. After typing the last name, press the end-of-file character (Ctrl-D) to signal the end of the list.
   ```
   WorkReport^D
   ```

After you press Ctrl-D, the system displays information similar to the following:
```
job joyce.741502800.a at Fri Jul  6 00:00:00 CDT 2002.
```

The program WorkReport is given the job number `joyce.741502800.a` and will run at midnight July 6.

To list the programs you have sent to be run later, type:
```
at -l
```

The system displays information similar to the following:
```
joyce.741502800.a       Fri Jul  6 00:00:00 CDT 2002
```

See the **at** command in the *AIX 5L Version 5.2 Commands Reference* for the exact syntax.

Listing All Scheduled Processes (at or atq Command)

You can list all scheduled processes by using the **-l** flag with the **at** command or with the **atq** command. Both commands give the same output, but the **atq** command can order the processes by the time the **at** command was issued and can display just the number of processes in the queue.

You can list all scheduled processes in the following ways:
- With the **at** command from the command line
- With the **atq** command

For user restrictions on using the **at** command, see the **Note** inScheduling a Process for Later Operation (at Command).

at Command

To list the scheduled processes, type:

```
at -l
```

This command lists all the scheduled processes in your queue. If you are a root user, this command lists all the scheduled processes for all users. For complete details of the syntax, see the **at** command.

atq Command

To list all scheduled processes in the queue, type:

```
atq
```

If you are a root user, you can list the scheduled processes in a particular user's queue by typing:

```
atq UserName
```

To list the number of scheduled processes in the queue, type:

```
atq -n
```

Removing a Process from the Schedule (at Command)

You can remove a scheduled process with the **at** command using the **-r** flag. For user restrictions on using the **at** command, see the **Note** inScheduling a Process for Later Operation (at Command).

From the Command Line

1. To remove a scheduled process, you must know the process number. You can obtain the process number using the **at -l** command or the **atq** command. See "Listing All Scheduled Processes (at or atq Command)" on page 41 for details.
2. When you know the number of the process you want to remove, type:

   ```
   at -r ProcessNumber
   ```

You can also use the **smit rmat** command to perform this task.

Removing a Background Process (kill Command)

If INTERRUPT does not halt your foreground process or 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 process using the **kill** command, you must know its PID number. The general format for the **kill** command is as follows:

```
kill ProcessID
```

> **Note:** To remove a process, you must have root user authority or be the user who started the process. The default signal to a process from the **kill** command is -15 (SIGTERM).

kill Command

Note: To remove a zombie process, you must remove its parent process.

1. Use the **ps** command to determine the process ID of the process you want to remove. You might want to pipe this command through a **grep** command to list only the process you want. For example, if you want the process ID of a vi session, you could type:

   ```
   ps -l | grep vi
   ```
2. In the following example, you issue the **find** command to run in the background. You then decide to cancel the process. Issue the **ps** command to list the PID numbers.

```
$ find / -type f > dir.paths &
[1]    21593
$ ps
   PID   TTY  TIME  COMMAND
  1627  pts3  0:00  ps
  5461  pts3  0:00  ksh
 17565  pts3  0:00  -ksh
 21593  pts3  0:00  find / -type f
$ kill 21593
$ ps
   PID   TTY  TIME  COMMAND
  1627  pts3  0:00  ps
  5461  pts3  0:00  ksh
 17565  pts3  0:00  -ksh
[1]  +  Terminated 21593     find  /  -type f > dir.paths  &
```

The command **kill 21593** ends the background **find** process, and the second **ps** command returns no status information about PID 21593. The system does not display the termination message until you enter your next command, unless that command is **cd**.

The **kill** command lets you cancel background processes. You might want to do this if you realize that you have mistakenly put a process in the background or a process is taking too long to run.

See the **kill** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

The **kill** command can also used in **smit** by typing:

```
smit kill
```

Command Summary for Commands and Processes

Commands

alias	Shell command that prints a list of aliases to standard output
history	Shell command that displays the history event list
man	Displays information about commands, subroutines, and files online
wsm	Performs system management from a web browser
whatis	Describes the function a command performs
whereis	Locates the source, binary, or manual for installed programs

Processes

at	Runs commands at a later time, lists all scheduled processes, or removes a process from the schedule
atq	Displays the queue of jobs waiting to be run
kill	Sends a signal to running processes
nice	Runs a command at a lower or higher priority.
ps	Shows current status of processes.
renice	Alters priority of running processes

Related Information

"Commands Overview" on page 26

"Processes Overview" on page 35

Chapter 12, "Shells" on page 139

“Korn Shell or POSIX Shell Commands” on page 144

“Bourne Shell” on page 184

“C Shell” on page 200

Chapter 5. Input and Output Redirection

The operating system allows you to manipulate the input and output (I/O) of data to and from your system by using specific I/O commands and symbols. You can control input by specifying the location from which to gather data. For example, you can specify to read input while data is entered on the keyboard (standard input) or to read input from a file. You can control output by specifying where to display or store data. You can specify to write output data to the screen (standard output) or to write it to a file.

The operating system, because it is multitasking, is designed to handle processes in combination with each other. This chapter discusses the advantages of redirecting input and output and of tying processes together.

This chapter discusses the following:

Standard Input, Standard Output, and Standard Error

When a command begins running, it usually expects that the following files are already open: standard input, standard output, and standard error (sometimes called *error output* or *diagnostic* output). A number, called a *file descriptor,* is associated with each of these files, as follows:

File descriptor 0	Standard input
File descriptor 1	Standard output
File descriptor 2	Standard error (diagnostic) output

A child process normally inherits these files from its parent. All three files are initially assigned to the workstation (0 to the keyboard, 1 and 2 to the display). The shell permits them to be redirected elsewhere before control is passed to a command.

When you enter a command, if no file name is given, your keyboard is the *standard input*, sometimes denoted as *stdin*. When a command finishes, the results are displayed on your screen.

Your screen is the *standard output*, sometimes denoted as *stdout*. By default, commands take input from the standard input and send the results to standard output.

Error messages are directed to standard error, sometimes denoted as *stderr*. By default, this is your screen.

These default actions of input and output can be varied. You can use a file as input and write results of a command to a file. This is called *input/output redirection*.

The output from a command, which normally goes to the display device, can easily be redirected to a file instead. This is known as *output redirection*. This is useful when you have a lot of output that is difficult to read on the screen or when you want to put files together to create a larger file.

Though not used as much as output redirection, the input for a command, which normally comes from the keyboard, can also be redirected from a file. This is known as *input redirection*. Redirection of input lets you prepare a file in advance and then have the command read the file.

Redirecting Standard Output

When the notation > *filename* is added to the end of a command, the output of the command is written to the specified file name. The > symbol is known as the *output redirection operator*.

Any command that outputs its results to the screen can have its output sent to a file.

Redirecting Output to a File

The output of a process can be redirected to a file by typing the command followed by the file name. For example, to send the results of the **who** command to a file called **users**, type:

```
who > users
```

Press Enter.

> **Note:** If the **users** file already exists, it is deleted and replaced, unless the **noclobber** option of the **set** built-in **ksh** (Korn shell) or **csh** (C shell) command is specified.

To see the contents of the **users** file , type:

```
cat users
```

Press Enter.

A list similar to the following displays:

```
denise    lft/0 May 13 08:05
marta     pts/1 May 13 08:10
endrica   pts/2 May 13 09:33
```

Redirecting Output and Appending to a File

When the notation >> *filename* is added to the end of a command, the output of the command is appended to the specified file name, rather than writing over any existing data. The >> symbol is known as the *append redirection operator*.

For example, to append **file2** to **file1**, type:

```
cat file2 >> file1
```

Press Enter.

> **Note:** If the **file1** file does not exist, it is created, unless the **noclobber** option of the **set** built-in **ksh** (Korn shell) or **csh** (C shell) command is specified.

Creating a Text File with Redirection from the Keyboard

Used alone, the **cat** command uses whatever you type at the keyboard as input. You can redirect this input to a file. Enter Ctrl-D on a new line to signal the end of the text.

At the system prompt, type:

```
cat > filename
This is a test.
^D
```

Concatenating Text Files

Combining various files into one file is known as *concatenation*.

For example, at the system prompt, type:

```
cat file1 file2 file3 > file4
```

Press Enter.

The previous example creates `file4`, which consists of `file1`, `file2`, and `file3`, appended in the order given.

The following example shows a common error when concatenating files:

```
cat file1 file2 file3 > file1
```

> **Attention:** In this example, you might expect the **cat** command to append the contents of `file1`, `file2`, and `file3` into `file1`. The **cat** command creates the output file first, so it actually erases the contents of `file1` and then appends `file2` and `file3` to it.

Redirecting Standard Input

When the notation < *filename* is added to the end of a command, the input of the command is read from the specified file name. The < symbol is known as the *input redirection operator*.

> **Note:** Only commands that normally take their input from the keyboard can have their input redirected.

For example, to send the file `letter1` as a message to user `denise` with the **mail** command, type:

```
mail denise < letter1
```

Press Enter.

Discarding Output with the /dev/null File

The **/dev/null** file is a special file. This file has a unique property; it is always empty. Any data you send to **/dev/null** is discarded. This is a useful feature when you run a program or command that generates output you want to ignore.

For example, you have a program named `myprog` that accepts input from the screen and generates messages while it is running that you would rather ignore. To read input from the file `myscript` and discard the standard output messages, type:

```
myprog < myscript >/dev/null
```

Press Enter.

In this example, `myprog` uses the file `myscript` as input, and all standard output is discarded.

Redirecting Standard Error and Other Output

In addition to the standard input and standard output, commands often produce other types of output, such as error or status messages known as diagnostic output. Like standard output, standard error output is written to the screen unless redirected.

If you want to redirect standard error or other output, you must use a file descriptor. A *file descriptor* is a number associated with each of the I/O files that a command ordinarily uses. File descriptors can also be specified to redirect standard input and standard output, but are already the default values. The following numbers are associated with standard input, output, and error:

0	Standard input (keyboard)
1	Standard output (display)
2	Standard error (display)

To redirect standard error output, type the file descriptor number 2 in front of the output or append redirection symbols (> or > >) and a file name after the symbol. For example, the following command takes the standard error output from the **cc** command where it is used to compile the **testfile.c** file and appends it to the end of the **ERRORS** file:

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

Other types of output can also be redirected using the file descriptors from 0 through 9. For example, if the **cmd** command writes output to file descriptor 9, you can redirect that output to the **savedata** file with the following command:

```
cmd 9> savedata
```

If a command writes to more than one output, you can independently redirect each one. Suppose that a command directs its standard output to file descriptor 1, directs its standard error output to file descriptor 2, and builds a data file on file descriptor 9. The following command line redirects each of these outputs to a different file:

```
command > standard 2> error 9> data
```

Using Inline Input (Here) Documents

If a command is in the following form:

```
command << eofstring
```

and *eofstring* is any string that does not contain pattern-matching characters, then the shell takes the subsequent lines as the standard input of *command* until the shell reads a line consisting of only *eofstring* (possibly preceded by one or more tab characters). The lines between the first *eofstring* and the second are frequently referred to as an *inline input*, or *here*, document. If a hyphen (-) immediately follows the << redirection characters, the shell strips leading tab characters from each line of the here document before it passes the line to the *command*.

The shell creates a temporary file containing the here document and performs variable and command substitution on the contents before passing the file to the command. It performs pattern matching on file names that are part of command lines in command substitutions. To prohibit all substitutions, quote any character of the *eofstring*:

```
command << \eofstring
```

The here document is especially useful for a small amount of input data that is more conveniently placed in the shell procedure rather than kept in a separate file (such as editor scripts). For instance, you could type:

```
cat <<- xyz
   This message will be shown on the
   display with leading tabs removed.
   xyz
```

Press Enter.

Using Pipes and Filters

You can connect two or more commands so that the standard output of one command is used as the standard input of another command. A set of commands connected this way is known as a *pipeline*. The connection that joins the commands is known as a *pipe*. Pipes are useful because they let you tie many single-purpose commands into one powerful command.

You can direct the output from one command to become the input for another command using a pipeline. The commands are connected by a pipe (|) symbol.

When a command takes its input from another command, modifies it, and sends its results to standard output, it is known as a *filter*. Filters can be used alone but they are especially useful in pipelines. The most common filters are as follows:

- sort
- more
- pg

For example, the **ls** command writes the contents of the current directory to the screen in one scrolling data stream. When more than one screen of information is presented, some data is lost from view. To control the output so the contents display screen by screen, you can use a pipeline to direct the output of the **ls** command to the **pg** command, which controls the format of output to the screen as shown in the following example:

```
ls | pg
```

In the example, the output of the **ls**command is the input for the **pg** command. Press Enter to continue to the next screen.

Pipelines operate in one direction only (left to right). Each command in a pipeline runs as a separate process and all processes 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.

Another example of using pipes is with the **grep** command. The **grep** command searches a file for lines that contain strings of a certain pattern. To display all your files created or modified in July, type:

```
ls -l | grep Jul
```

Press Enter.

In the example, the output of the **ls** command is the input for the **grep** command.

Displaying Program Output and Copying to a File (tee command)

The **tee** command, used with a pipe, reads standard input, then writes the output of a program to standard output and simultaneously copies it into the specified file or files. Use the **tee** command to view your output immediately and at the same time, store it for future use.

For example, type:

```
ps -ef | tee program.ps
```

Press Enter.

This displays the standard output of the **ps -ef** command at the display device, and at the same time saves a copy of it in the **program.ps** file. If the **program.ps** file already exists, it is deleted and replaced, unless the **noclobber** option of the **set** built-in command is specified.

For example, to view and save the output from a command to an existing file:

```
ls -l | tee -a program.ls
```

This displays the standard output of **ls -l** at the display device and at the same time appends a copy of it to the end of the **program.ls** file.

The system displays information similar to the following, and the **program.ls** file contains the same information:

```
-rw-rw-rw-   1 jones   staff   2301   Sep 19    08:53 161414
-rw-rw-rw-   1 jones   staff   6317   Aug 31    13:17 def.rpt
-rw-rw-rw-   1 jones   staff   5550   Sep 10    14:13 try.doc
```

See the **tee** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Clearing Your Screen (clear Command)

You can empty the screen of messages and keyboard input with the **clear** command.

At the prompt, type:

```
clear
```

Press Enter.

The system clears the screen and displays the prompt.

Sending a Message to Standard Output (echo Command)

You can display messages on the screen with the **echo** command.

For example, to write a message to standard output, at the prompt, type:

```
echo Please insert diskette . . .
```

Press Enter.

The system displays the following:

```
Please insert diskette . . .
```

For example, to use the **echo** command with pattern-matching characters, at the prompt, type:

```
echo The back-up files are: *.bak
```

Press Enter.

The system displays the message `The back-up files are:` followed by the file names in the current directory ending with `.bak`.

Appending a Single Line of Text to a File (echo Command)

You can add a single line of text to a file with the **echo** command, used with the append redirection operator.

For example, at the prompt, type:

```
echo Remember to backup mail files by end of week.>

>notes
```

Press Enter.

This adds the message `Remember to backup mail files by end of week.` to the end of the file `notes`.

Copying Your Screen to a File (capture and script Commands)

You can copy everything printed on your terminal to a file that you specify with the **capture** command, which emulates a VT100 terminal.

You can use the **script** command to copy everything printed on your terminal to a file that you specify, without emulating a VT100 terminal.

Both commands are useful for printing records of terminal dialogs.

For example, to capture the screen of a terminal while emulating a VT100, at the prompt, type:

```
capture screen.01
```

Press Enter.

The system displays information similar to the following:

```
Capture command is started. The file is screen.01.
Use ^P to dump screen to file screen.01.
You are now emulating a vt100 terminal.
Press Any Key to continue.
```

After entering data and dumping the screen contents, stop the **capture** command by pressing Ctrl-D or typing `exit` and pressing Enter. The system displays information similar to the following:

```
Capture command is complete. The file is screen.01.
You are NO LONGER emulating a vt100 terminal.
```

Use the **cat** command to display the contents of your file.

For example, to capture the screen of a terminal, at the prompt, type:

```
script
```

Press Enter.

The system displays information similar to the following:

```
Script command is started. The file is typescript.
```

Everything displayed on the screen is now copied to the **typescript** file.

To stop the **script** command, press Ctrl-D or type `exit` and press Enter. The system displays information similar to the following:

```
Script command is complete. The file is typescript.
```

Use the **cat** command to display the contents of your file.

See the **capture** and **script** commands in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Displaying Text in Large Letters on Your Screen (banner Command)

The **banner** command displays ASCII characters to your screen in large letters. Each line in the output can be up to 10 digits (or uppercase or lowercase characters) in length.

For example, at the prompt, type:

```
banner GOODBYE!
```

Press Enter.

The system displays `GOODBYE!` in large letters on your screen.

Command Summary for Input and Output Redirection

>	"Redirecting Standard Output" on page 46
<	"Redirecting Standard Input" on page 47
> >	"Redirecting Output and Appending to a File" on page 46
|	"Using Pipes and Filters" on page 49
banner	Writes ASCII character strings in large letters to standard output
capture	Allows terminal screens to be dumped to a file
clear	Clears the terminal screen
echo	Writes character strings to standard output
script	Allows terminal input and output to be copied to a file
tee	Displays the standard output of a program and copies it into a file

Related Information

"Commands Overview" on page 26

"Processes Overview" on page 35

Chapter 12, "Shells" on page 139

"Korn Shell or POSIX Shell Commands" on page 144

"Bourne Shell" on page 184

"C Shell" on page 200

Chapter 7, "Files" on page 67

Chapter 6. File Systems and Directories

File systems consist of groups of directories and the files within the directories. File systems are commonly represented as an inverted tree. The root directory, symbolized by the slash (/) symbol, defines a file system and appears at the top of a file system tree diagram. Directories branch downward from the root directory in the tree diagram and can contain bothfiles and subdirectories. Branching creates unique paths through the directory structure to every object in the file system.

Collections of files are stored in *directories*. These collections of files are often related to each other; storing them in a structure of directories keeps them organized.

A *file* is a collection of data that can be read from or written to. A file can be a program you create, text you write, data you acquire, or a device you use. Commands, printers, terminals, correspondence, and application programs are all stored in files. This allows users to access diverse elements of the system in a uniform way and gives great flexibility to the file system.

This chapter discusses the following:

File Systems

A *file system* is a hierarchical structure (file tree) of files and directories. This type of structure resembles an inverted tree with the roots at the top and the branches at the bottom. This file tree uses directories to organize data and programs into groups, allowing the management of many directories and files at one time.

Some tasks are performed more efficiently on a file system than on each directory within the file system. For example, you can back up, move, or secure an entire file system.

The basic type of file system is called the *Journaled File System (JFS)*. This file system uses database journaling techniques to maintain its structural consistency. This prevents damage to the file system when the system is halted abnormally.

Some of the most important system management tasks have to do with file systems, specifically:
- Allocating space for file systems on logical volumes
- Creating file systems
- Making file system space available to system users
- Monitoring file system space usage
- Backing up file systems to guard against data loss if the system fails
- Maintaining file systems in a consistent state

These tasks should be performed by your system administrator.

File System Types

The operating system supports multiple file system types. These include:

Journaled File System (JFS)	The basic file system type, it supports the entire set of file system commands.
Enhanced Journaled File System (JFS2)	The basic file system type, it supports the entire set of file system commands.
Network File System (NFS)	A file system type that permits files residing on remote machines to be accessed as though they reside on the local machine.
CD-ROM File System (CDRFS)	A file system type that allows the contents of a CD-ROM to be accessed through the normal file system interfaces (open, read, and close).

File System Structure

On standalone machines, the following file systems reside on the associated devices by default:

/**File System**	/**Device**
/**dev**/**hd1**	/**home**
/**dev**/**hd2**	**/usr**
/**dev**/**hd3**	/**tmp**
/**dev**/**hd4**	/**(root)**
/**dev**/**hd9var**	/**var**
/**proc**	/**proc**
/**dev**/**hd10opt**	/**opt**

The file tree has the following characteristics:
- Files that can be shared by machines of the same hardware architecture are located in the **/usr** file system.
- Variable per-client files, for example, spool and mail files, are located in the **/var** file system.
- The **/(root)** file system contains files and directories critical for system operation. For example, it contains
  - A device directory (**/dev**)
  - Mount points where file systems can be mounted onto the root file system, for example, **/mnt**

- The **/home** file system is the mount point for users' home directories.
- For servers, the **/export** directory contains paging-space files, per-client (unshared) root file systems, dump, home, and **/usr/share** directories for diskless clients, as well as exported **/usr** directories.
- The **/proc** file system contains information about the state of processes and threads in the system.
- The **/opt** file system contains optional software, such as applications.

The following list provides information about the contents of some of the subdirectories of the **/(root)** file system.

/bin	Symbolic link to the **/usr/bin** directory.
/dev	Contains device nodes for special files for local devices. The **/dev** directory contains special files for tape drives, printers, disk partitions, and terminals.
/etc	Contains configuration files that vary for each machine. Examples include: • **/etc/hosts** • **/etc/passwd**
/export	Contains the directories and files on a server that are for remote clients.
/home	Serves as a mount point for a file system containing user home directories. The **/home** file system contains per-user files and directories. In a standalone machine, a separate local file system is mounted over the **/home** directory. In a network, a server might contain user files that should be accessible from several machines. In this case, the server's copy of the **/home** directory is remotely mounted onto a local **/home** file system.
/lib	Symbolic link to the **/usr/lib** directory, which contains architecture-independent libraries with names in the form **lib*.a**.
/sbin	Contains files needed to boot the machine and mount the **/usr** file system. Most of the commands used during booting come from the boot image's RAM disk file system; therefore, very few commands reside in the **/sbin** directory.
/tmp	Serves as a mount point for a file system that contains system-generated temporary files.
/u	Symbolic link to the **/home** directory.
/usr	Serves as a mount point for a file system containing files that do not change and can be shared by machines (such as executable programs and ASCII documentation). Standalone machines mount a separate local file system over the **/usr** directory. Diskless and disk-poor machines mount a directory from a remote server over the **/usr** file system.
/var	Serves as a mount point for files that vary on each machine. The **/var** file system is configured as a file system because the files that it contains tend to grow. For example, it is a symbolic link to the **/usr/tmp** directory, which contains temporary work files.

Displaying Available Space on a File System (df Command)

You can use the **df** command to display information about total space and available space on a file system. The *FileSystem* parameter specifies the name of the device on which the file system resides, the directory on which the file system is mounted, or the relative path name of a file system. If you do not specify the *FileSystem* parameter, the **df** command displays information for all currently mounted file systems. If a file or directory is specified, then the **df** command displays information for the file system on which it resides.

Normally, the **df** command uses free counts contained in the superblock. Under certain exceptional conditions, these counts might be in error. For example, if a file system is being actively modified when the **df** command is running, the free count might not be accurate.

See the **df** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Note: On some remote file systems, such as Network File Systems (NFS), the columns representing the available space on the display are left blank if the server does not provide the information.

The following are examples of how to use the **df** command:

1. To display information about all mounted file systems, type:

   ```
   df
   ```

   Press Enter.

   If your system is configured so the **/**, **/usr**, **/site**, and **/usr/venus** directories reside in separate file systems, the output from the **df** command is similar to the following:

   ```
   Filesystem 512-blocks  free   %used   Iused %Iused   Mounted on
   /dev/hd4    20480     13780    32%      805   13%     /
   /dev/hd2   385024     15772    95%    27715   28%     /usr
   /dev/hd9var 40960     38988     4%      115    1%     /var
   /dev/hd3    20480     18972     7%       81    1%     /tmp
   /dev/hd1     4096      3724     9%       44    4%     /home
   ```

2. To display available space on the file system in which your current directory resides, type:

   ```
   df .
   ```

   Press Enter.

Directory Overview

A *directory* is a unique type of file that contains only the information needed to access files or other directories. As a result, a directory occupies less space than other types of files. Directories enable you to group files and other directories, allowing you to organize the file system into a modular hierarchy and giving the file-system structure flexablilty and depth. Unlike other types of files, a special set of commands control directories.

Directories contain directory entries. Each entry contains a file or subdirectory name and an index node reference number (*i-node* number). To increase speed and enhance use of disk space, the data in a file is stored at various locations in the memory of the computer. The i-node number contains the addresses used to locate all the scattered blocks of data associated with a file. The i-node number also records other information about the file, including time of modification and access, access modes, number of links, file owner, and file type. It is possible to link several names for a file to the same i-node number by creating directory entries with the **ln** command.

Because directories often contain information that should not be available to all users of the system, directory access can be protected. By setting a directory's permissions, you can control who has access to the directory, also determining which users (if any) can alter information within the directory. See "File and Directory Access Modes" on page 119 for more information.

This section discusses:

- "Types of Directories"
- "Directory Organization" on page 57
- "Directory Naming Conventions" on page 57
- "Directory Path Names" on page 57
- "Directory Abbreviations" on page 58
- "Directory-Handling Procedures" on page 58

Types of Directories

Directories can be defined by the operating system, the system administrator, or users. The system-defined directories contain specific kinds of system files, such as commands. At the top of the file system hierarchy is the system-defined **/(root)** directory. The **/(root)** directory usually contains the following standard system-related directories:

/dev	Contains special files for I/O devices.
/etc	Contains files for system initialization and system management.
/home	Contains login directories for the system users.
/tmp	Contains files that are temporary and can be deleted in a specified number of days.
/usr	Contains the **lpp**, **include**, and other system directories.
/usr/bin	Contains user-executable programs.

Some directories, such as your login or home directory (**$HOME**), are defined and customized by the system administrator. When you log in to the operating system, the login directory is the current directory.

Directories that you create are called user-defined directories. These directories help you organize and maintain your files.

Directory Organization

Directories contain files, subdirectories, or a combination of both. A subdirectory is a directory within a directory. The directory containing the subdirectory is called the *parent directory*.

For the operating system to track and find directories, each directory has an entry for the parent directory in which it was created, .. (dot dot), and an entry for the directory itself, . (dot). In most directory listings, these files are hidden.

Directory Tree
The file system structure of directories can easily become complex. Attempt to keep the file and directory structure as simple as possible. Also, create files and directories with easily recognizable names. This makes working with files easier.

Parent Directory
Each directory, except for **/(root)**, has one parent directory and can have zero or more child directories.

Home Directory
When you log in, the system puts you in a directory called your *home directory* or login directory. This directory is set up by the system administrator for each user. Your home directory is the repository for your personal files. Normally, directories that you create for your own use will be subdirectories of your home directory. To return to your home directory at any time, type the **cd** command and press Enter at the prompt.

Working Directory
You are always working within a directory. Whichever directory you are currently working in is called your *current* or *working* directory. The **pwd** (present working directory) command reports the name of your working directory. Use the **cd** command to change working directories.

Directory Naming Conventions

The name of each directory must be unique within the directory where it is stored. This ensures that the directory also has a unique path name in the file system. Directories follow the same naming conventions that files do, as explained in “File-Naming Conventions” on page 69.

Directory Path Names

Each file and directory can be reached by a unique path, known as the *path name*, through the file system tree structure. The path name specifies the location of a directory or file within the file system.

Note: Path names cannot exceed 1023 characters in length.

The file system uses the following kinds of path names:

absolute path name	Traces the path from the **/(root)** directory. Absolute path names always begin with the slash (/) symbol.
relative path name	Traces the path from the current directory through its parent or its subdirectories and files.

An absolute path name represents the complete name of a directory or file from the **/(root)** directory downward. Regardless of where you are working in the file system, you can always find a directory or file by specifying its absolute path name. Absolute path names start with a slash (/), the symbol representing the root directory. The path name **/A/D/9** is the absolute path name for **9**. The first slash (/) represents the **/(root)** directory, which is the starting place for the search. The remainder of the path name directs the search to **A**, then to **D,** and finally to **9**.

Two files named **9** can exist because the absolute path names to the files give each file a unique name within the file system. The path names **/A/D/9** and **/C/E/G/9** specify two unique files named **9**.

Unlike full path names, relative path names specify a directory or file based on the current working directory. For relative path names, you can use the notation dot dot (..) to move upward in the file system hierarchy. The dot dot (..) represents the parent directory. Because relative path names specify a path starting in the current directory, they do not begin with a slash (/). Relative path names are used to specify the name of a file in the current directory or the path name of a file or directory above or below the level of the current directory in the file system. If **D** is the current directory, the relative path name for accessing **10** is **F/10**, but the absolute path name is always **/A/D/F/10**. Also, the relative path name for accessing 3 is **../../B/3**.

You can also represent the name of the current directory by using the notation dot (.). The dot (.) notation is commonly used when running programs that read the current directory name.

Directory Abbreviations

Abbreviations provide a convenient way to specifycertain directories. The following is a list of abbreviations.

Abbreviation	**Meaning**
.	The current working directory.
..	The directory above the current working directory (the parent directory).
~	Your home directory (this is not true for the Bourne shell. For more information, see “Bourne Shell” on page 184).
$HOME	Your home directory (this is true for all shells).

Directory-Handling Procedures

You can work with directories and their contents in a variety of ways.

The command and an example are presented for each of the following directory tasks:

- “Creating a Directory (mkdir Command)” on page 59
- “Moving or Renaming a Directory (mvdir Command)” on page 59
- “Displaying the Current Directory (pwd Command)” on page 60
- “Changing to Another Directory (cd Command)” on page 60
- “Copying a Directory (cp Command)” on page 61
- “Displaying Contents of a Directory (ls Command)” on page 61
- “Deleting or Removing a Directory (rmdir Command)” on page 63

- “Comparing the Contents of Directories (dircmp Command)” on page 63

Creating a Directory (mkdir Command)

You can use the **mkdir** command to create one or more directories specified by the *Directory* parameter. Each new directory contains the standard entries dot (.) and dot dot (..). You can specify the permissions for the new directories with the **-m** *Mode* flag.

When you create a directory, it is created within the current, or working, directory unless you specify an absolute path name to another location in the file system.

The following are examples of how to us the **mkdir** command:

1. To create a new directory called **Test** in the current working directory with default permissions, type:
   ```
   mkdir Test
   ```
   Press Enter.
2. To create a directory called **Test** with `rwxr-xr-x` permissions in a previously created **/home/demo/sub1** directory, type:
   ```
   mkdir -m 755 /home/demo/sub1/Test
   ```
   Press Enter.
3. To create a directory called **Test** with default permissions in the **/home/demo/sub2** directory, type:
   ```
   mkdir -p /home/demo/sub2/Test
   ```
   Press Enter.

   The **-p** flag creates the **/home**, **/home/demo**, and **/home/demo/sub2** directories if they do not already exist.

See the **mkdir** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Moving or Renaming a Directory (mvdir Command)

To move or rename a directory, use the **mvdir** command.

For example, to move a directory, type:
```
mvdir book manual
```
Press Enter.

This moves the **book** directory under the directory named **manual**, if the **manual** directory exists. Otherwise, the **book** directory is renamed to **manual**.

For example, to move and rename a directory, type:
```
mvdir book3 proj4/manual
```
Press Enter.

This moves the **book3** directory to the directory named **proj4** and renames **proj4** to **manual** (if the **manual** directory did not previously exist).

See the **mvdir** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Displaying the Current Directory (pwd Command)

You can use the **pwd** command to write to standard output the full path name of your current directory (from the **/(root)** directory). All directories are separated by a slash (/). The **/(root)** directory is represented by the first slash (/), and the last directory named is your current directory.

For example, to display your current directory, type:

```
pwd
```

Press Enter.

The full path name of your current directory displays similar to the following:

```
/home/thomas
```

Changing to Another Directory (cd Command)

The **cd** command moves you from your present directory to another directory. You must have execute (search) permission in the specified directory.

If you do not specify a *Directory* parameter, the **cd** command moves you to your login directory (**$HOME** in the **ksh** and **bsh** environments, or **$home** in the **csh** environment). If the specified directory name is a full path name, it becomes the current directory. A full path name begins with a slash (/) indicating the **/(root)** directory, a dot (.) indicating current directory, or a dot dot (..) indicating parent directory. If the directory name is not a full path name, the **cd** command searches for it relative to one of the paths specified by the **$CDPATH** shell variable (or **$cdpath csh** variable). This variable has the same syntax as, and similar semantics to, the **$PATH** shell variable (or **$path csh** variable).

The following are examples of how to use the **cd** command:

1. To change to your home directory, type:

   ```
   cd
   ```

   Press Enter.
2. To change to the **/usr/include** directory, type:

   ```
   cd /usr/include
   ```

   Press Enter.
3. To go down one level of the directory tree to the **sys** directory, type:

   ```
   cd sys
   ```

   Press Enter.

   If the current directory is **/usr/include** and it contains a subdirectory named **sys**, then **/usr/include/sys** becomes the current directory.
4. To go up one level of the directory tree, type:

   ```
   cd ..
   ```

   Press Enter.

   The special file name, dot dot (..), refers to the directory immediately above the current directory, its parent directory.

See the **cd** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Copying a Directory (cp Command)

You can use the **cp** command to create a copy of the contents of the file or directory specified by the *SourceFile* or *SourceDirectory* parameters into the file or directory specified by the *TargetFile* or *TargetDirectory* parameters. If the file specified as the *TargetFile* exists, the copy writes over the original contents of the file. If you are copying more than one *SourceFile*, the target must be a directory.

To place a copy of the *SourceFile* into a directory, specify a path to an existing directory for the *TargetDirectory* parameter. Files maintain their respective names when copied to a directory unless you specify a new file name at the end of the path. The **cp** command also copies entire directories into other directories if you specify the **-r** or **-R** flags.

The following are examples of how to use the **cp** command.

1. To copy all the files in the **/home/accounts/customers/orders** directory to the **/home/accounts/customers/shipments** directory, type:
   ```
   cp /home/accounts/customers/orders/* /home/accounts/customers/shipments
   ```
   Press Enter.

   This copies only the files in **orders** directory into the **shipments** directory.
2. To copy a directory, including all its files and subdirectories, to another directory, type:
   ```
   cp -R /home/accounts/customers /home/accounts/vendors
   ```
   Press Enter.

   This copies the **customers** directory, including all its files, subdirectories, and the files in those subdirectories, into the **vendors** directory.

See the **cp** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Displaying Contents of a Directory (ls Command)

You can display the contents of a directory by using the **ls** command.

ls command

The **ls** command writes to standard output the contents of each specified *Directory* or the name of each specified *File*, along with any other information you ask for with the flags. If you do not specify a *File* or *Directory*, the **ls** command displays the contents of the current directory.

By default, the **ls** command displays all information in alphabetic order by file name. If the command is executed by a user with root authority, it uses the **-A** flag by default, listing all entries except dot (.) and dot dot (..). To show all entries for files, including those that begin with a . (dot), use the **ls -a** command.

You can format the output in the following ways:

- List one entry per line, using the **-l** flag.
- List entries in multiple columns, by specifying either the **-C** or **-x** flag. The **-C** flag is the default format when output is to a tty.
- List entries in a comma-separated series, by specifying the **-m** flag.

To determine the number of character positions in the output line, the **ls** command uses the **$COLUMNS** environment variable. If this variable is not set, the command reads the **terminfo** file. If the **ls** command cannot determine the number of character positions by either of these methods, it uses a default value of 80.

The information displayed with the **-e** and **-l** flags is interpreted as follows:

The first charactor may be one of the following:

d	Entry is a directory.
b	Entry is a block special file.
c	Entry is a character special file.
l	Entry is a symbolic link.
p	Entry is a first-in, first-out (FIFO) pipe special file.
s	Entry is a local socket.
-	Entry is an ordinary file.

The next nine characters are divided into three sets of three characters each. The first three characters show the owner's permission. The next set of three characters shows the permission of the other users in the group. The last set of three characters shows the permission of anyone else with access to the file. The three characters in each set show read, write, and execute permission of the file. Execute permission of a directory lets you search a directory for a specified file.

Permissions are indicated as follows:

r	Read permission granted
t	Only the directory owner or the file owner can delete or rename a file within that directory, even if others have write permission to the directory.
w	Write (edit) permission granted
x	Execute (search) permission granted
-	Corresponding permission not granted.

The information displayed with the **-e** flag is the same as with the **-l** flag, except for the addition of an 11th character, interpreted as follows:

+	Indicates a file has extended security information. For example, the file might have extended **ACL**, **TCB**, or **TP** attributes in the mode.
-	Indicates a file does not have extended security information.

When the size of the files in a directory are listed, the **ls** command displays a total count of blocks, including indirect blocks.

For example, to list all files in the current directory, type:

```
ls -a
```

Press Enter.

This lists all files, including

- dot (.)
- dot dot (..)
- Other files whose names might or might not begin with a dot (.)

For example, to display detailed information, type:

```
ls -l chap1 .profile
```

Press Enter.

This displays a long listing with detailed information about **chap1** and **.profile**.

For example, to display detailed information about a directory, type:

```
ls -d -l . manual manual/chap1
```

Press Enter.

This displays a long listing for the directories **.** and **manual**, and for the file **manual/chap1**. Without the **-d** flag, this would list the files in the **.** and **manual** directories instead of the detailed information about the directories themselves.

See the **ls** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Deleting or Removing a Directory (rmdir Command)

You can use the **rmdir** command to remove the directory, specified by the *Directory* parameter, from the system. The directory must be empty (it can only contain . and ..) before you can remove it, and you must have write permission in its parent directory. Use the **ls -a** *Directory* command to check whether the directory is empty.

The following are examples of how to use the **rmdir** command:

1. To empty and remove a directory, type:

   ```
   rm mydir/* mydir/.*
   rmdir mydir
   ```

   Press Enter.

   This removes the contents of **mydir**, then removes the empty directory. The **rm** command displays an error message about trying to remove the directories dot (.) and dot dot (..), and then the **rmdir** command removes them and the directory itself.

   Note: **rm mydir/* mydir/.*** first removes files with names that do not begin with a dot, and then removes those with names that do begin with a dot. You might not realize that the directory contains file names that begin with a dot because the **ls** command does not normally list them unless you use the **-a** flag.
2. To remove the **/tmp/jones/demo/mydir** directory and all the directories beneath it, type:

   ```
   cd /tmp
   rmdir -p jones/demo/mydir
   ```

   Press Enter.

   This removes the **jones/demo/mydir** directory from the **/tmp** directory. If a directory is not empty or you do not have write permission to it when it is to be removed, the command terminates with appropriate error messages.

See the **rmdir** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Comparing the Contents of Directories (dircmp Command)

You can use the **dircmp** command to compare two directories specified by the *Directory1* and *Directory2* parameters and write information about their contents to standard output. First, the **dircmp** command compares the file names in each directory. If the same file name is contained in both, the **dircmp** command compares the contents of both files.

In the output, the **dircmp** command lists the files unique to each directory. It then lists the files with identical names in both directories, but with different contents. If no flag is specified, it also lists files that have identical contents as well as identical names in both directories.

The following are examples of how to use the **dircmp** command:

1. To summarize the differences between the files in the **proj.ver1** and **proj.ver2** directories, type:

   ```
   dircmp proj.ver1 proj.ver2
   ```

   Press Enter.

   This displays a summary of the differences between the **proj.ver1** and **proj.ver2** directories. The summary lists separately the files found only in one directory or the other, and those found in both. If a file is found in both directories, the **dircmp** command notes whether the two copies are identical.
2. To show the details of the differences between the files in the **proj.ver1** and **proj.ver2** directories, type:

   ```
   dircmp -d -s proj.ver1 proj.ver2
   ```

   Press Enter.

   The **-s** flag suppresses information about identical files. The **-d** flag displays a **diff** listing for each of the differing files found in both directories.

See the **dircmp** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Command Summary for File Systems and Directories

File Systems

df	Reports information about space on file systems.

Directory Abbreviations

.	The current working directory.
..	The directory above the current working directory (the parent directory).
~	Your home directory (this is not true for the Bourne shell. For more information, see “Bourne Shell” on page 184).
$HOME	Your home directory (this is true for all shells).

Directory Handling Procedures

cd	Changes the current directory.
cp	Copies files or directories.
dircmp	Compares two directories and the contents of their common files.
ls	Displays the contents of a directory.
mkdir	Creates one or more new directories.
mvdir	Moves (renames) a directory.
pwd	Displays the path name of the working directory.
rmdir	Removes a directory.

Related Information

“Commands Overview” on page 26

“Processes Overview” on page 35

Chapter 5, “Input and Output Redirection” on page 45

"File Systems" on page 53

"Directory Overview" on page 56

Chapter 7, "Files" on page 67

"Linking Files and Directories" on page 82

Chapter 9, "Backup Files and Storage Media" on page 103

Chapter 10, "File and System Security" on page 117

Chapter 7. Files

Files are used for all input and output (I/O) of information in the operating system. To standardize access to both software and hardware. Input occurs when the contents of a file is modified or written to. Output occurs when the contents of one file is read or transferred to another file. For example, to create a printed copy of a file, the system reads the information from the text file and writes that information to the file representing the printer.

This chapter discusses the following:

Types of Files

The following basic types of files exist:

regular	Stores data (text, binary, and executable)
directory	Contains information used to access other files
special	Defines a FIFO (first-in, first-out) pipe file or a physical device

All file types recognized by the system fall into one of these categories. However, the operating system uses many variations of these basic types.

Regular Files

Regular files are the most common files and are used to contain data. Regular files are in the form of text files or binary files:

Text Files

Text files are regular files that contain information stored in ASCII and readable by the user. You can display and print these files. The lines of a text file must not contain **NUL** characters, and none can exceed **{LINE_MAX}** bytes in length, including the new-line character.

The term *text file* does not prevent the inclusion of control or other nonprintable characters (other than **NUL**). Therefore, standard utilities that list text files as inputs or outputs are either able to process the special characters or they explicitly describe their limitations within their individual sections.

Binary Files

Binary files are regular files that contain information readable by the computer. Binary files might be executable files that instruct the system to accomplish a job. Commands and programs are stored in executable, binary files. Special compiling programs translate ASCII text into binary code.

Text and binary files differ only in that text files have lines of less than **{LINE_MAX}** bytes, with no **NUL** characters, each terminated by a newline character.

Directory Files

Directory files contain information that the system needs to access all types of files, but directory files do not contain the actual file data. As a result, directories occupy less space than a regular file and give the file system structure flexibility and depth. Each directory entry represents either a file or a subdirectory. Each entry contains the name of the file and the file's index node reference number (i-node number). The i-node number points to the unique index node assigned to the file. The i-node number describes the location of the data associated with the file. Directories are created and controlled by a separate set of commands.

Special Files

Special files define devices for the system or temporary files created by processes. The basic types of special files are FIFO (first-in, first-out), block, and character. FIFO files are also called *pipes*. Pipes are created by one process to temporarily allow communication with another process. These files cease to exist when the first process finishes. Block and character files define devices.

Every file has a set of permissions (called *access modes*) that determine who can read, modify, or execute the file.

To learn more about file access modes, see "File and Directory Access Modes" on page 119 .

File-Naming Conventions

The name of each file must be unique within the directory where it is stored. This ensures that the file also has a unique path name in the file system. File-naming guidelines are:

- A file name can be up to 255 characters long and can contain letters, numbers, and underscores.
- The operating system is case-sensitive, which means it distinguishes between uppercase and lowercase letters in file names. Therefore, `FILEA`, `FiLea`, and `filea` are three distinct file names, even if they reside in the same directory.
- File names should be as descriptive and meaningful as possible.
- Directories follow the same naming conventions as files.
- Certain characters have special meaning to the operating system. Avoid using these charactors when you are naming files. These characters include the following:
  ```
  / \ " ' * ; - ? [ ] ( ) ~ ! $ { } < > # @ & |
  ```
- A file name is hidden from a normal directory listing if it begins with a dot (.). When the **ls** command is entered with the **-a** flag, the hidden files are listed along with regular files and directories.

File Path Names

The path name for each file and directory in the file system consists of the names of every directory that precedes it in the tree structure.

Because all paths in a file system originate from the /(**root**) directory, each file in the file system has a unique relationship to the root directory, known as the *absolute path name*. Absolute path names begin with the slash (/) symbol. For example, the absolute path name of file **h** could be**/B/C/h**. Notice that two files named **h** can exist in the system. Because the absolute paths to the two files are different, **/B/h** and **/B/C/h**, each file named **h** has a unique name within the system. Every component of a path name is a directory except the final component. The final component of a path name can be a file name.

Note: Path names cannot exceed 1023 characters in length.

Pattern Matching with Wildcards and Metacharacters

Wildcard characters provide a convenient way for specifying multiple file or directory names with one character. The wildcard characters are asterisk (*) and question mark (?). The metacharacters are open and close square brackets ([]), hyphen (-), and exclamation mark (!).

Using the * Wildcard Charactor

Use the asterisk (*) to match any sequence or string of characters. The (*) indicates any characters, including no characters. For example, if you have the following files in your directory:

```
1test 2test afile1 afile2 bfile1 file file1 file10 file2 file3
```

and you want to refer to only the files that begin with `file`, you would use:

```
file*
```

The files selected would be: `file file1 file10 file2 file3`

To refer to only the files that contain the word `file`, you would use:

```
*file*
```

The files selected would be: `afile1 afile2 bfile1 file file1 file10 file2 file3`

Using the ? Wildcard Charactor

Use the ? to match any one character. The ? indicates any single character.

To refer to only the files that start with `file` and end with a single character, use:

```
file?
```

The files selected would be: `file1 file2 file3`

To refer to only the files that start with `file` and end with any two characters, use:

```
file??
```

The file selected would be: `file10`

Using [ ] Shell Metacharacters

Metacharacters offer another type of wildcard notation by enclosing the desired characters within [ ]. It is like using the ?, but it allows you to choose specific characters to be matched. The [ ] also allow you to specify a range of values using the hyphen (-). To specify all the letters in the alphabet, use [[:alpha:]]. To specify all the lowercase letters in the alphabet, use [[:lower:]].

To refer to only the files that end in `1` or `2`, use:

```
*file[12]
```

The files selected would be: `afile1 afile2 file1 file2`

To refer to only the files that start with any number, use:

`[0123456789]*` **or** `[0-9]*`

The files selected would be: `1test 2test`

To refer to only the files that do not begin with an `a`, use:

```
[!a]*
```

The files selected would be: `1test 2test bfile1 file file1 file10 file2 file3`

Pattern Matching versus Regular Expressions

Regular expressions allow you to select specific strings from a set of character strings. The use of regular expressions is generally associated with text processing.

Regular expressions can represent a wide variety of possible strings. While many regular expressions can be interpreted differently depending on the current locale, internationalization features provide for contextual invariance across locales.

See the examples in the following comparison between File Matching Patterns and Regular Expressions:

Pattern Matching	**Regular Expression**
`*`	`.*`
`?`	`.`
`[!a]`	`[^a]`
`[abc]`	`[abc]`
`[[:alpha:]]`	`[[:alpha:]]`

See the **awk** command in the *AIX 5L Version 5.2 Commands Reference* for the exact syntax.

File Handling Procedures

There are many ways to work with the files on your system. Usually you create a text file with a text editor. The common editors in the UNIX environment are **vi** and **ed**. Because several text editors are available, you can choose to edit with the editor you feel comfortable with.

You can also create files by using input and output redirection, as described in ″Chapter 5, “Input and Output Redirection” on page 45″ . You can send the output of a command to a new file or append it to an existing file.

After creating and modifying files, you might have to copy or move files from one directory to another, rename files to distinguish different versions of a file, or give different names to the same file. You might also need to create directories when working on different projects.

Also, you might need to delete certain files. Your directory can quickly get cluttered with files that contain old or useless information. To release storage space on your system, ensure that you delete files that are no longer needed.

This section discusses the following:
- “Deleting Files (rm Command)”
- “Moving and Renaming Files (mv Command)” on page 72
- “Copying Files (cp Command)” on page 72
- “Finding Files (find Command)” on page 74
- “Displaying the File Type (file Command)” on page 75
- “Displaying File Contents (pg, more, page, and cat Commands)” on page 75
- “Finding Text Strings Within Files (grep Command)” on page 76
- “Sorting Text Files (sort Command)” on page 77
- “Comparing Files (diff Command)” on page 78
- “Counting Words, Lines, and Bytes in Files (wc Command)” on page 78
- “Displaying the First Lines of Files (head Command)” on page 79
- “Displaying the Last Lines of Files (tail Command)” on page 79
- “Cutting Sections of Text Files (cut Command)” on page 80
- “Pasting Sections of Text Files (paste Command)” on page 80
- “Numbering Lines in Text Files (nl Command)” on page 81
- “Removing Columns in Text Files (colrm Command)” on page 81

Deleting Files (rm Command)

When you no longer need a file, you can remove it with the **rm** command. The **rm** command removes the entries for a specified file, group of files, or certain select files from a list within a directory. User confirmation, read permission, and write permission are not required before a file is removed when you use the **rm** command. However, you must have write permission for the directory containing that file.

The following are examples of how to use the **rm** command:

1. To delete the file named **myfile**, type:
   ```
   rm myfile
   ```
   Press Enter.
2. To delete all the files in the **mydir** directory, one by one, type:
   ```
   rm -i mydir/*
   ```

Press Enter.

After each file name displays, type `y` and press Enter to delete the file. Or to keep the file, just press Enter.

See the **rm** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Moving and Renaming Files (mv Command)

To move files and directories from one directory to another or rename a file or directory, use the **mv** command. If you move a file or directory to a new directory without specifying a new name, it retains its original name.

Attention: The **mv** command can overwrite many existing files unless you specify the **-i** flag. The **-i** flag prompts you to confirm before it overwrites a file. The **-f** flag does not prompt you. If both the **-f** and **-i** flags are specified in combination, the last flag specified takes precedence.

Moving Files with mv Command

The following are examples of how to use the **mv** command:

1. To move a file to another directory and give it a new name, type:

   ```
   mv intro manual/chap1
   ```

   Press Enter.

   This moves the **intro**file to the **manual/chap1** directory. The name **intro** is removed from the current directory, and the same file appears as **chap1** in the **manual** directory.
2. To move a file to another directory, keeping the same name, type:

   ```
   mv chap3 manual
   ```

   Press Enter.

   This moves **chap3** to **manual/chap3**.

Renaming Files with mv Command

You can use the **mv** command to change the name of a file without moving it to another directory.

To rename a file, type:

```
mv appendix apndx.a
```

Press Enter.

This renames the **appendix**file to **apndx.a**. If a file named **apndx.a** already exists, its old contents are replaced with those of the **appendix** file.

See the **mv** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Copying Files (cp Command)

You can use the **cp** command to create a copy of the contents of the file or directory specified by the *SourceFile* or *SourceDirectory* parameters into the file or directory specified by the *TargetFile* or *TargetDirectory* parameters. If the file specified as the *TargetFile* exists, the copy writes over the original contents of the file without warning. If you are copying more than one *SourceFile*, the target must be a directory.

If a file with the same name exists at the new destination, the copied file overwrites the file at the new destination. Therefore, it is a good practice to assign a *new* name for the copy of the file to ensure that a file of the same name does not exist in the destination directory.

To place a copy of the *SourceFile* into a directory, specify a path to an existing directory for the *TargetDirectory* parameter. Files maintain their respective names when copied to a directory unless you specify a new file name at the end of the path. The **cp** command also copies entire directories into other directories if you specify the **-r** or **-R** flags.

You can also copy special-device files using the **-R** flag. Specifying **-R** causes the special files to be re-created under the new path name. Specifying the **-r** flag causes the **cp** command to attempt to copy the special files to regular files.

The following are examples of how to use the **cp** command:

1. To make a copy of a file in the current directory, type:
   ```
   cp prog.c prog.bak
   ```
   Press Enter.

   This copies **prog.c** to **prog.bak**. If the **prog.bak** file does not already exist, then the **cp** command creates it. If it does exist, then the **cp** command replaces it with a copy of the **prog.c** file.
2. To copy a file in your current directory into another directory, type:
   ```
   cp jones /home/nick/clients
   ```
   Press Enter.

   This copies the **jones** file to **/home/nick/clients/jones**.
3. To copy all the files in a directory to a new directory, type:
   ```
   cp /home/janet/clients/* /home/nick/customers
   ```
   Press Enter.

   This copies only the files in the **clients** directory to the **customers** directory.
4. To copy a specific set of files to another directory, type:
   ```
   cp jones lewis smith /home/nick/clients
   ```
   Press Enter.

   This copies the **jones**, **lewis**, and **smith** files in your current working directory to the **/home/nick/clients** directory.
5. To use pattern-matching characters to copy files, type:
   ```
   cp programs/*.c .
   ```
   Press Enter.

   This copies the files in the **programs** directory that end with `.c` to the current directory, indicated by the single dot (.). You must type a space between the `c` and the final dot.

See the **cp** command in the *AIX 5L Version 5.2 Commands Reference* for the exact syntax.

Finding Files (find Command)

You can use the **find** command to recursively searche the directory tree for each specified *Path*, seeking files that match a Boolean expression written using the terms given in the following text. The output from the **find** command depends on the terms specified by the *Expression* parameter.

The following are examples of how to use the **find** command:

1. To list all files in the file system with the name **.profile**, type:
   ```
   find / -name .profile
   ```
   Press Enter.

   This searches the entire file system and writes the complete path names of all files named `.profile`. The slash (/) tells the **find** command to search the /(**root**) directory and all of its subdirectories.

   To save time, limit the search by specifying the directories where you think the files might be.
2. To list files having a specific permission code of **0600** in the current directory tree, type:
   ```
   find . -perm 0600
   ```
   Press Enter.

   This lists the names of the files that have *only* owner-read and owner-write permission. The dot (.) tells the **find** command to search the current directory and its subdirectories. For an explanation of permission codes, see the **chmod** command.
3. To search several directories for files with certain permission codes, type:
   ```
   find manual clients proposals -perm -0600
   ```
   Press Enter.

   This lists the names of the files that have owner-read and owner-write permission and possibly other permissions. The **manual**, **clients**, and **proposals** directories and their subdirectories are searched. In the previous example, **-perm 0600** selects only files with permission codes that match **0600** exactly. In this example, **-perm -0600** selects files with permission codes that allow the accesses indicated by **0600** and other accesses above the **0600** level. This also matches the permission codes 0622 and 2744.
4. To list all files in the current directory that have been changed during the current 24-hour period, type:
   ```
   find . -ctime 1
   ```
   Press Enter.
5. To search for regular files with multiple links, type:
   ```
   find . -type f -links +1
   ```
   Press Enter.

   This lists the names of the ordinary files (`-type f`) that have more than one link (`-links +1`).

   Note: Every directory has at least two links: the entry in its parent directory and its own **.**(dot) entry. For more information on multiple file links, see the **ln** command.
6. To search for all files that are exactly 414 bytes in length, type:
   ```
   find . -size 414c
   ```
   Press Enter.

See the **find** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Displaying the File Type (file Command)

You can use the **file** command to read the files specified by the *File* or **-f** *FileList* parameter, perform a series of tests on each one. The command attempt to classify the files by type. The command then writes the file types to standard output.

If a file appears to be ASCII, the **file** command examines the first 512 bytes and determines its language. If a file does not appear to be ASCII, the **file** command further attempts to determine whether it is a binary data file or a text file that contains extended characters.

If the *File* parameter specifies an executable or object module file and the version number is greater than 0, the **file** command displays the version stamp.

The **file** command uses the **/etc/magic** file to identify files that have a magic number, that is, any file containing a numeric or string constant that indicates the type.

The following are examples of how to use the **file** command:

1. To display the type of information the file named **myfile** contains, type:
   ```
   file myfile
   ```
   Press Enter.

   This displays the file type of **myfile** (such as directory, data, ASCII text, C-program source, and archive).
2. To display the type of each file named in the **filenames.lst**file, which contains a list of file names, type:
   ```
   file -f filenames.lst
   ```
   Press Enter.

   This displays the type of each file named in the **filenames.lst** file. Each file name must display on a separate line.
3. To create the **filenames.lst** file, so that it contains all the file names in the current directory, type:
   ```
   ls > filenames.lst
   ```
   Press Enter.

   Edit the **filenames** file as desired.

See the **file** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Displaying File Contents (pg, more, page, and cat Commands)

The **pg**, **more**, and **page** commands allow you to view the contents of a file and control the speed at which your files are displayed. You can also use the **cat** command to display the contents of one or more files on your screen. Combining the **cat** command with the **pg** command allows you to read the contents of a file one full screen at a time.

You can also display the contents of files by using input and output redirection. See Chapter 5, “Input and Output Redirection” on page 45 for more details on input and output redirection.

pg Command

The **pg** command reads the file names from the *File* parameter and writes them to standard output one screen at a time. If you specify hyphen (-) as the *File* parameter, or run the **pg** command without options, the **pg** command reads standard input. Each screen is followed by a prompt. If you press the Enter key, another screen displays. Subcommands used with the **pg** command let you review something that has already passed.

For example, to look at the contents of the file `myfile` one page at a time, type:

```
pg myfile
```

Press Enter.

See the **pg** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

more or page Command

The **more** or **page** command displays continuous text one screen at a time. It pauses after each screen and prints the *filename* and percent completed (for example, `myfile (7%)`) at the bottom of the screen. If you then press the Enter key, the **more** command displays an additional line. If you press the spacebar, the **more** command displays another screen of text.

> **Note:** On some terminal models, the **more** command clears the screen, instead of scrolling, before displaying the next screen of text.

For example, to view a file named `myfile`, type:

```
more myfile
```

Press Enter.

Press the spacebar to view the next screen.

See the **more** command in the*AIX 5L Version 5.2 Commands Reference* for the complete syntax.

cat Command

The **cat** command reads each *File* parameter in sequence and writes it to standard output.

For example, to display the contents of the file `notes`, type:

```
cat notes
```

Press Enter. If the file is more than 24 lines long, some of it scrolls off the screen. To list a file one page at a time, use the **pg** command.

For example, to display the contents of the files `notes`, `notes2`, and `notes3`, type:

```
cat notes notes2 notes3
```

Press Enter.

See the **cat** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Finding Text Strings Within Files (grep Command)

The **grep** command searches for the pattern specified by the *Pattern* parameter and writes each matching line to standard output.

The following are examples of how to use the **grep** command:

1. To search in a file named **pgm.s** for a pattern that contains some of the pattern-matching characters *, ^, ?, [,], \(, \), \{, and \}, in this case, lines starting with any lowercase or uppercase letter, type:

   ```
   grep "^[a-zA-Z]" pgm.s
   ```

   Press Enter.

   This displays all lines in the **pgm.s** file that begin with a letter.
2. To display all lines in a file named **sort.c** that do not match a pattern, type:

   ```
   grep -v bubble sort.c
   ```

   Press Enter.

   This displays all lines that do not contain the word **bubble** in the **sort.c** file.
3. To display lines in the output of the **ls** command that match the string **staff**, type:

   ```
   ls -l | grep staff
   ```

   Press Enter.

See the **grep** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Sorting Text Files (sort Command)

You can use the **sort** command to alphabetize or sequence lines in the files specified by the *File* parameters and writes the result to standard output. If the *File* parameter specifies more than one file, the **sort** command concatenates the files and alphabetizes them as one file.

Note: The **sort** command is case-sensitive and orders uppercase letters before lowercase (this is dependent on the locale).

In the following examples, the contents of the file named **names** are:

```
marta
denise
joyce
endrica
melanie
```

and the contents of the file named **states** are:

```
texas
colorado
ohio
```

1. To display the sorted contents of the file named **names**, type:

   ```
   sort names
   ```

   Press Enter.

   The system displays information similar to the following:

   ```
   denise
   endrica
   joyce
   marta
   melanie
   ```
2. To display the sorted contents of the **names** and **states** files, type:

   ```
   sort names states
   ```

Press Enter.

The system displays information similar to the following:

```
colorado
denise
endrica
joyce
marta
melanie
ohio
texas
```

3. To replace the original contents of the file named **names** with its sorted contents, type:

   ```
   sort -o names names
   ```

   Press Enter.

   This replaces the contents of the **names** file with the same data but in sorted order.

See the **sort** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Comparing Files (diff Command)

You can use the **diff** command to compare text files. It can compare single files or the contents of directories.

When the **diff** command is run on regular files, and when it compares text files in different directories, the **diff** command tells which lines must be changed in the files so that they match.

The following are examples of how to use the **diff** command:

1. To compare two files, type:

   ```
   diff chap1.bak chap1
   ```

   Press Enter.

   This displays the differences between the **chap1.bak** and **chap1** files.
2. To compare two files while ignoring differences in the amount of white space, type:

   ```
   diff -w prog.c.bak prog.c
   ```

   Press Enter. If the two files differ only in the number of spaces and tabs between words, the **diff -w** command considers the files to be the same.

See the **diff** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Counting Words, Lines, and Bytes in Files (wc Command)

By default, the **wc** command counts the number of lines, words, and bytes in the files specified by the *File* parameter. If a file is not specified for the *File* parameter, standard input is used. The command writes the results to standard output and keeps a total count for all named files. If flags are specified, the ordering of the flags determines the ordering of the output. A *word* is defined as a string of characters delimited by spaces, tabs, or newline characters.

When files are specified on the command line, their names are printed along with the counts.

For example, to display the line, word, and byte counts of the file named `chap1`, type:

```
wc chap1
```

Press Enter. This displays the number of lines, words, and bytes in the `chap1` file.

For example, to display only byte and word counts, type:

```
wc -cw chap*
```

Press Enter. This displays the number of bytes and words in each file where the name starts with `chap`, and displays the totals.

See the **wc** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Displaying the First Lines of Files (head Command)

The **head** command writes to standard output the first few lines of each of the specified files or of the standard input. If no flag is specified with the **head** command, the first 10 lines are displayed by default.

For example, to display the first five lines of the `Test` file, type:

```
head -5 Test
```

Press Enter.

See the **head** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Displaying the Last Lines of Files (tail Command)

The **tail** command writes the file specified by the *File* parameter to standard output beginning at a specified point.

For example, to display the last 10 lines of the `notes` file, type:

```
tail notes
```

Press Enter.

For example, to specify the number of lines to start reading from the end of the `notes` file, type:

```
tail -20 notes
```

Press Enter.

For example, to display the `notes` file one page at a time, beginning with the 200th byte, type:

```
tail -c +200 notes | pg
```

Press Enter.

For example, to follow the growth of the file named `accounts`, type:

```
tail -f accounts
```

Press Enter. This displays the last 10 lines of the `accounts` file. The **tail** command continues to display lines as they are added to the `accounts` file. The display continues until you press the (Ctrl-C) key sequence to stop the display.

See the **tail** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Cutting Sections of Text Files (cut Command)

To write selected bytes, characters, or fields from each line of a file to standard output, use the **cut** command.

For example, to display several fields of each line of a file, type:

```
cut -f1,5 -d: /etc/passwd
```

Press Enter. This displays the login name and full user name fields of the system password file. These are the first and fifth fields (`-f1,5`) separated by colons (`-d:`).

For example, if the **/etc/passwd** file looks like this:

```
su:*:0:0:User with special privileges:/:/usr/bin/sh
daemon:*:1:1::/etc:
bin:*:2:2::/usr/bin:
sys:*:3:3::/usr/src:
adm:*:4:4:System Administrator:/var/adm:/usr/bin/sh
pierre:*:200:200:Pierre Harper:/home/pierre:/usr/bin/sh
joan:*:202:200:Joan Brown:/home/joan:/usr/bin/sh
```

the **cut** command produces:

```
su:User with special privileges
daemon:
bin:
sys:
adm:System Administrator
pierre:Pierre Harper
joan:Joan Brown
```

See the **cut** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Pasting Sections of Text Files (paste Command)

The **paste** command merges the lines of up to 12 files into one file.

For example, if you have a file named `names` that contains the following text:

```
rachel
jerry
mark
linda
scott
```

another file named `places` that contains the following text:

```
New York
Austin
Chicago
Boca Raton
Seattle
```

and another file named `dates` that contains the following text:

```
February 5
March 13
June 21
July 16
November 4
```

To paste the text of the files `names`, `places`, and `dates` together, type:

```
paste names places dates > npd
```

Press Enter. This creates a file named `npd` that contains the data from the `names` file in one column, the `places` file in another, and the `dates` file in a third. The `npd` file now contains the following:

```
rachel          New York        February 5
jerry           Austin          March 13
mark            Chicago         June 21
linda           Boca Raton      July 16
scott           Seattle         November 4
```

A tab character separates the name, place, and date on each line. These columns do not align, because the tab stops are set at every eighth column.

For example, to separate the columns with a character other than a tab, type:

```
paste -d"!@" names places dates > npd
```

Press Enter. This alternates ! and @ as the column separators. If the `names`, `places`, and `dates` files are the same as in example 1, then the `npd` file contains the following:

```
rachel!New York@February 5
jerry!Austin@March 13
mark!Chicago@June 21
linda!Boca Raton@July 16
scott!Seattle@November 4
```

For example, to list the current directory in four columns, type:

```
ls | paste - - - -
```

Press Enter. Each hyphen (-) tells the **paste** command to create a column containing data read from the standard input. The first line is put in the first column, the second line in the second column, and so on.

See the **paste** command in the *AIX 5L Version 5.2 Commands Reference* for the exact syntax.

Numbering Lines in Text Files (nl Command)

The **nl** command reads the specified file (standard input by default), numbers the lines in the input, and writes the numbered lines to standard output.

For example, to number only the nonblank lines, type:

```
nl chap1
```

Press Enter. This displays a numbered listing of `chap1`, numbering only the nonblank lines in the body sections.

For example, to number all lines, type:

```
nl -ba chap1
```

Press Enter. This numbers all the lines in the file named `chap1`, including blank lines.

See the **nl** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Removing Columns in Text Files (colrm Command)

The **colrm** command removes specified columns from a file. Input is taken from standard input. Output is sent to standard output.

If the command is called with one parameter, the columns of each line from the specified column to the last column are removed. If the command is called with two parameters, the columns from the first specified column to the second specified column are removed.

Note: Column numbering starts with column 1.

For example, to remove columns from the `text.fil` file, type:

```
colrm 6 < text.fil
```

Press Enter.

If `text.fil` contains:

```
123456789
```

then the **colrm** command displays:

```
12345
```

See the **colrm** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Linking Files and Directories

Links are connections between a file name and an index node reference number (i-node number), the internal representation of a file. Because directory entries contain file names paired with i-node numbers, every directory entry is a link. The i-node number actually identifies the file, not the file name. By using links, any i-node number or file can be known by many different names.

For example, i-node number 798 contains a memo regarding June sales in the Omaha office. Presently, the directory entry for this memo is as follows:

i-node Number	File Name
798	**memo**

Because this information relates to information stored in the `sales` and `omaha` directories, linking is used to share the information where it is needed. Using the **ln** command, links are created to these directories. Now the file has three file names as follows:

i-node Number	File Name
798	**memo**
798	**sales/june**
798	**omaha/junesales**

When you use the **pg** or **cat** command to view the contents of any of the three file names, the same information is displayed. If you edit the contents of the i-node number from any of the three file names, the contents of the data displayed by all of the file names will reflect any changes.

Types of Links

Links are created with the **ln** command and are of the following types:

hard link	Allows access to the data of a file from a new file name. Hard links ensure the existence of a file. When the last hard link is removed, the i-node number and its data are deleted. Hard links can be created only between files that are in the same file system.
symbolic link	Allows access to data in other file systems from a new file name. The symbolic link is a special type of file that contains a path name. When a process encounters a symbolic link, the process may search that path. Symbolic links do not protect a file from deletion from the file system.

Note: The user who creates a file retains ownership of that file no matter how many links are created. Only the owner of the file or the root user can set the access mode for that file. However, changes can be made to the file from a linked file name with the proper access mode.

A file or directory exists as long as there is one hard link to the i-node number for that file. In the long listing displayed by the **ls -l** command, the number of hard links to each file and subdirectory is given. All hard links are treated equally by the operating system regardless of which link was created first.

Linking Files (ln Command)

Linking files with the **ln** command is a convenient way to work with the same data in more than one place. Links are created by giving alternate names to the original file. The use of links allows a large file, such as a database or mailing list, to be shared by several users without making copies of that file. Not only do links save disk space, but changes made to one file are automatically reflected in all the linked files.

The **ln** command links the file designated in the *SourceFile* parameter to the file designated by the *TargetFile* parameter or to the same file name in another directory specified by the *TargetDirectory* parameter. By default, the **ln** command creates hard links. To use the **ln** command to create symbolic links, designate the **-s** flag.

If you are linking a file to a new name, you can list only one file. If you are linking to a directory, you can list more than one file.

The *TargetFile* parameter is optional. If you do not designate a target file, the **ln** command creates a file in your current directory. The new file inherits the name of the file designated in the *SourceFile* parameter.

Note: You cannot link files across file systems without using the **-s** flag.

For example, to create another link to a file named `chap1`, type:

```
ln -f chap1 intro
```

Press Enter. This links `chap1` to the new name, `intro`. When the **-f** flag is used, the file name `intro` is created if it does not already exist. If `intro` does exist, the file is replaced by a link to `chap1`. Then both the `chap1` and `intro` file names will refer to the same file. Any changes made to one file also appear in the other.

For example, to link a file named `index` to the same name in another directory named `manual`, type:

```
ln index manual
```

Press Enter. This links `index` to the new name, `manual/index`.

For example, to link several files to names in another directory, type:

```
ln chap2 jim/chap3 /home/manual
```

Press Enter. This links `chap2` to the new name `/home/manual/chap2` and `jim/chap3` to `/home/manual/chap3`.

For example, to use the **ln** command with pattern-matching characters, type:

```
ln manual/* .
```

Note: You must type a space between the asterisk and the period.

Press Enter. This links all files in the `manual` directory into the current directory, dot (.), giving them the same names they have in the `manual` directory.

For example, to create a symbolic link, type:

```
ln -s /tmp/toc toc
```

Press Enter. This creates the symbolic link, `toc`, in the current directory. The `toc` file points to the `/tmp/toc` file. If the `/tmp/toc` file exists, the **cat** `toc` command lists its contents.

To achieve identical results without designating the *TargetFile* parameter, type:

```
ln -s /tmp/toc
```

Press Enter.

See the **ln** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Removing Linked Files

The **rm** command removes the link from the file name that you indicate. When one of several hard-linked file names is deleted, the file is not completely deleted because it remains under the other name. When the last link to an i-node number is removed, the data is removed as well. The i-node number is then available for reuse by the system.

See the **rm** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

DOS Files

The AIX operating system allows you to work with DOS files on your system. Copy to a diskette the DOS files you want to work with. Your system can read these files into a base operating system directory in the correct format and back onto the diskette in DOS format.

> **Note:** The wildcard characters * and ? (asterisk and question mark) do not work correctly with the commands discussed in this section (although they do with the base operating system shell). If you do not specify a file name extension, the file name is matched as if you had specified a blank extension.

Copying DOS Files to Base Operating System Files

The **dosread** command copies the specified DOS file to the specified base operating system file.

> **Note:** DOS file-naming conventions are used with one exception. Because the backslash (\) character can have special meaning to the base operating system, use a slash (/) character as the delimiter to specify subdirectory names in a DOS path name.

For example, to copy a text file named `chap1.doc` from a DOS diskette to the base operating file system, type:

```
dosread -a chap1.doc chap1
```

Press Enter. This copies the DOS text file **\CHAP1.DOC** on the **/dev/fd0** default device to the base operating system file **chap1** in the current directory.

For example, to copy a binary file from a DOS diskette to the base operating file system, type:

```
dosread -D/dev/fd0 /survey/test.dta /home/fran/testdata
```

Press Enter. This copies the **\SURVEY\TEST.DTA**DOS data file on **/dev/fd1** to the base operating system file `/home/fran/testdata`.

See the **dosread** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Copying Base Operating System Files to DOS Files

The **doswrite** command copies the specified base operating system file to the specified DOS file.

> **Note:** DOS file-naming conventions are used with one exception. Because the backslash (\) character can have special meaning to the base operating system, use a slash (/) character as the delimiter to specify subdirectory names in a DOS path name.

For example, to copy a text file named `chap1` from the base operating file system to a DOS diskette, type:

```
doswrite -a chap1 chap1.doc
```

Press Enter. This copies the base operating system file `chap1` in the current directory to the DOS text file `\CHAP1.DOC` on **/dev/fd0**.

For example, to copy a binary file named `/survey/test.dta` from the base operating file system to a DOS diskette, type:

```
doswrite -D/dev/fd0 /home/fran/testdata /survey/test.dta
```

Press Enter. This copies the base operating system data file `/home/fran/testdata` to the DOS file `\SURVEY\TEST.DTA` on **/dev/fd1**.

See the **doswrite** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Deleting DOS Files

The **dosdel** command deletes the specified DOS file.

> **Note:** DOS file-naming conventions are used with one exception. Because the backslash (\) character can have special meaning to the base operating system, use a slash (/) character as the delimiter to specify subdirectory names in a DOS path name.

The **dosdel** command converts lowercase characters in the file or directory name to uppercase before it checks the disk. Because all file names are assumed to be full (not relative) path names, you need not add the initial slash (/).

For example, to delete a DOS file named `file.ext` on the default device (**/dev/fd0**), type:

```
dosdel file.ext
```

Press Enter.

See the **dosdel** command in the *AIX 5L Version 5.2 Commands Reference* for the completet syntax.

Listing Contents of a DOS Directory

The **dosdir** command displays information about the specified DOS files or directories.

> **Note:** DOS file-naming conventions are used with one exception. Because the backslash (\) character can have special meaning to the base operating system, use a slash (/) character as the delimiter to specify subdirectory names in a DOS path name.

The **dosdir** command converts lowercase characters in the file or directory name to uppercase before it checks the disk. Because all file names are assumed to be full (not relative) path names, you need not add the initial / (slash).

For example, to read a directory of the DOS files on **/dev/fd0**, type:

```
dosdir
```

Press Enter. The command returns the names of the files and disk-space information, similar to the following.

```
PG3-25.TXT
PG4-25.TXT
PG5-25.TXT
PG6-25.TXT
Free space: 312320 bytes
```

See the **dosdir** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Command Summary for Files

*****	Wildcard, matches any characters.
?	Wildcard, matches any single character.
[ ]	Metacharacters, matches enclosed characters.

File-Handling Procedures

cat	Concatenates or displays files
cmp	Compares two files
colrm	Extracts columns from a file
cp	Copies files
cut	Writes out selected bytes, characters, or fields from each line of a file
diff	Compares text files
file	Determines the file type
find	Finds files with a matching expression
grep	Searches a file for a pattern
head	Displays the first few lines or bytes of a file or files
more	Displays continuous text one screen at a time on a display screen
mv	Moves files
nl	Numbers lines in a file
pg	Formats files to the display
rm	Removes (unlinks) files or directories
paste	Merges the lines of several files or subsequent lines in one file
page	Displays continuous text one screen at a time on a display screen
sort	Sorts files, merges files that are already sorted, and checks files to determine if they have been sorted
tail	Writes a file to standard output, beginning at a specified point
wc	Counts the number of lines, words, and bytes in a file

Linking Files and Directories

ln	Links files and directories

DOS Files

dosdel	Deletes DOS files
dosdir	Lists the directory for DOS files
dosread	Copies DOS files to Base Operating System files
doswrite	Copies Base Operating System files to DOS files

Related Information

“Commands Overview” on page 26

“Processes Overview” on page 35

Chapter 5, “Input and Output Redirection” on page 45

Chapter 12, “Shells” on page 139

“File Systems” on page 53

“Directory Overview” on page 56

Chapter 7, “Files” on page 67

“Linking Files and Directories” on page 82

Chapter 8, “Printers, Print Jobs, and Queues” on page 89

Chapter 9, “Backup Files and Storage Media” on page 103

Chapter 10, “File and System Security” on page 117

Chapter 8. Printers, Print Jobs, and Queues

Depending on the printer, you can control the appearance and characteristics of the final output. The printers need not be located in the same area as the system unit and the system console. A printer can be attached directly to a local system, or a print job can be sent over a network to a remote system.

To handle print jobs with maximum efficiency, the system places each job into a queue to await printer availability. The system can save output from one or more files in the queue. As the printer produces the output from one file, the system processes the next job in the queue. This process continues until each job in the queue has been printed.

For detailed information about printers, print jobs, and queues, see the *AIX 5L Version 5.2 Guide to Printers and Printing*.

This chapter discusses the following chapters:

Printer Terminology

The following describes terms commonly used with printing.

Local Printers

When a printer is attached to a node or host, the printer is referred to as a *local printer*.

Print Job

A *print job* is a unit of work to be run on a printer. A print job can consist of printing one or more files, depending on how the print job is requested. The system assigns a unique job number to each job it runs.

Print Spooler

The *spooler* used for printing is not specifically a print job spooler. Instead, it provides a generic spooling function that can be used for queuing various types of jobs, including print jobs queued to a printer.

The spooler does not normally know what type of job it is queuing. When the system administrator defines a spooler queue, the purpose of the queue is defined by the spooler backend program that is specified for the queue. For example, if the spooler backend program is the **piobe** command (the printer I/O backend), the queue is a print queue. Likewise, if the spooler backend program is a compiler, the queue is for compile jobs. When the spooler's **qdaemon** command selects a job from a spooler queue, it runs the job by invoking the backend program specified by the system administrator when the queue was defined.

The main spooler command is the **enq** command. Although you can invoke this command directly to queue a print job, the following front-end commands are defined for submitting a print job: the **lp**, **lpr**, and **qprt** commands. A print request issued by one of these commands is first passed to the **enq** program, which then places the information about the file in the queue for the **qdaemon** to process.

Printer Backend

The *printer backend* is a collection of programs called by the spooler's **qdaemon** command to manage a print job that is queued for printing. The printer backend performs the following functions:

- Receives from the **qdaemon** command a list of one or more files to be printed
- Uses printer and formatting attribute values from the database, overridden by flags entered on the command line
- Initializes the printer before printing a file
- Runs filters as necessary to convert the print-data stream to a format supported by the printer
- Provides filters for simple formatting of ASCII documents
- Provides support for printing national language characters
- Passes the filtered print-data stream to the printer device driver
- Generates header and trailer pages
- Generates multiple copies
- Reports paper out, intervention required, and printer error conditions
- Reports problems detected by the filters
- Cleans up after a print job is canceled
- Provides a print environment that a system administrator can customize to address specific printing needs

qdaemon

The **qdaemon** is a process that runs in the background and controls the queues. It is generally started when the system is turned on.

Queue

The *queue* is where you direct a print job. It is a stanza in the **/etc/qconfig** file whose name is the name of the queue and points to the associated queue device. The following is a sample listing:

```
Msa1:
   device = lp0
```

In the previous example, `Msa1` is the queue name, and `lp0` is the device name.

Queue Device

The *queue device* is the stanza in the **/etc/qconfig** file that normally follows the local queue stanza. It specifies the **/dev** file (printer device) that should be printed to and the backend that should be used. Following is a sample listing:

```
lp0:
   file = /dev/lp0
   header = never
   trailer = never
   access = both
   backend = /usr/lpd/piobe
```

In the previous output, `lp0` is the device name and the rest of the lines define how the device is used.

Note: There can be more than one queue device associated with a single queue.

Real Printer

A *real printer* is the printer hardware attached to a serial or parallel port at a unique hardware device address. The printer device driver in the kernel communicates with the printer hardware and provides an interface between the printer hardware and a virtual printer, but it is not aware of the concept of virtual printers.

Remote Printers

A *remote print system* allows nodes that are not directly linked to a printer to have printer access. To use remote printing facilities, the individual nodes must be connected to a network using the Transmission Control Protocol/Internet Protocol (TCP/IP) and must support the required TCP/IP applications.

Virtual Printer

A *virtual printer* is a set of attributes that define a specific software view of a real printer. This view of the virtual printer refers only to the high-level data stream (such as ASCII or PostScript) that the printer understands. It does not include any information about how the printer hardware is attached to the host computer or about the protocol used for transferring bytes of data to and from the printer. Virtual printers are defined by the system manager.

Starting a Print Job (qprt Command)

To request a print job, use the **qprt** Command or the **smit** Command. For more information, see“Using the qprt Command” on page 92 and “Using the smit Command” on page 94. When using these commands, specify the following:

- Name of the file to print
- Print queue name
- Name of the output bin
- Number of copies to print
- Whether to make a copy of the file on the remote host
- Whether to erase the file after printing
- Whether to send notification of the job status
- Whether to send notification of the job status by the system mail
- Burst status
- User name for ″Delivery To″ label
- Console acknowledgment message for remote print
- File acknowledgment message for remote print
- Priority level

Prerequisites

Before yu start a print job, ensure the following:

- For local print jobs, the printer must be physically attached to your system.
- For remote print jobs, your system must be configured to communicate with the remote print server.

Using the qprt Command

The **qprt** command creates and queues a print job to print the file you specify. If you specify more than one file, all the files together make up one print job. These files are printed in the order specified on the command line.

Before you can print a file, you must have read access to it. To remove a file after it has printed, you must have write access to the directory that contains the file.

The most commonly used flags of the **qprt** command is as follows:

-b *Number*	Specifies the bottom margin. The bottom margin is the number of blank lines to be left at the bottom of each page.
-B *Value*	Specifies whether burst pages (continuous-form pages separated at perforations) should be printed. The *Value* variable consists of a two-character string. The first character applies to header pages. The second character applies to trailer pages. Each of the two characters can be one of the following: **a** Always prints the (header or trailer) page for each file in each print job. **n** Never prints the (header or trailer) page. **g** Prints the (header or trailer) page once for each print job (group of files). For example, the **-B ga** flag specifies that a header page be printed at the beginning of each print job and that a trailer page be printed after each file in each print job. **Note:** In a remote print environment, the default is determined by the remote queue on the server.
-e *Option*	Specifies whether emphasized print is wanted. **+** Indicates emphasized print is wanted. **!** Indicates emphasized print is not wanted.
-E *Option*	Specifies whether double-high print is wanted. **+** Indicates double-high print is wanted. **!** Indicates double-high print is not wanted.
-f *FilterType*	A one-character identifier that specifies a filter through which your print file or files are to be passed before being sent to the printer. The available filter identifiers are **p**, which invokes the **pr** filter, and **n**, which processes output from the **troff** command.
-i *Number*	Causes each line to be indented the specified number of spaces. The *Number* variable must be included in the page width specified by the **-w** flag.
-K *Option*	Specifies whether condensed print is wanted. **+** Indicates condensed print is wanted. **!** Indicates condensed print is not wanted.
-l *Number*	Sets the page length to the specified number of lines. If the *Number* variable is 0, page length is ignored, and the output is considered to be one continuous page. The page length includes the top and bottom margins and indicates the printable length of the paper.
-L *Option*	Specifies whether lines wider than the page width should be wrapped to the next line or truncated at the right margin. **+** Indicates that long lines should wrap to the next line. **!** Indicates that long lines should not wrap but instead should be truncated at the right margin.
-N *Number*	Specifies the number of copies to be printed. If this flag is not specified, one copy is printed.

-p *Number*	Sets the pitch to *Number* characters per inch. Typical values for *Number* are 10 and 12. The actual pitch of the characters printed is also affected by the values for the **-K** (condensed) flag and the **-W** (double-wide) flag.
-P *Queue*[:*QueueDevice*]	Specifies the print queue name and the optional queue device name. If this flag is not specified, the default printer is assumed.
-Q *Value*	Specifies paper size for the print job. The *Value* for paper size is printer-dependent. Typical values are: 1 for letter-size paper, 2 for legal, and so on. Consult your printer manual for the values assigned to specific paper sizes.
-t *Number*	Specifies the top margin. The top margin is the number of blank lines to be left at the top of each page.
-w *Number*	Sets the page width to the number of characters specified by the *Number* variable. The page width must include the number of indention spaces specified with the **-i** flag.
-W *Option*	Specifies whether double-wide print is wanted.

+	Indicates double-wide print is wanted.
!	Indicates double-wide print is not wanted.

-z *Value*	Rotates page printer output the number of quarter-turns clockwise as specified by the *Value* variable. The length (**-l**) and width (**-w**) values are automatically adjusted accordingly.

0	Portrait
1	Landscape right
2	Portrait upside-down
3	Landscape left

-= *OutputBin*	Specifies the output bin destination for a print job. The possible values are listed below. However, the valid output bins are printer-dependent.

0	Top printer bin
1-49	High Capacity Output (HCO) bins 1 - 49
49	Printer-specific output bins

-# *Value*	Specifies a special function.

j	Displays the job number for the specified print job.
h	Queues the print job, but puts it in the **HELD** state until it is released again.
v	Validates the specified printer backend flag values. This validation is useful in checking for illegal flag values at the time of submitting a print job. If the validation is not specified, an incorrect flag value will stop the print job later when the job is actually being processed.

For example, to request the **myfile** file to be printed on the first available printer configured for the default print queue using default values, type:

```
qprt myfile
```

For example, to request the file **somefile** to be printed on a specific queue using specific flag values and to validate the flag values at the time of print job submission, type:

```
qprt -f p -e + -Pfastest -# v somefile
```

This passes the **somefile** file through the **pr** filter command (the **-f p** flag) and prints it using emphasized mode (the **-e +** flag) on the first available printer configured for the queue named **fastest** (the **-Pfastest** flag).

For example, to print **myfile** on legal-size paper, type:

```
qprt -Q2 myfile
```

For example, to print three copies of each of the **new.index.c**, **print.index.c**, and **more.c** files at the print queue **Msp1**, type:

```
qprt -PMsp1 -N 3 new.index.c print.index.c more.c
```

For example, to print three copies of the concatenation of the files `new.index.c`, `print.index.c`, and `more.c`, type:

```
cat new.index.c print.index.c more.c | qprt -PMsp1 -N 3
```

> **Note:** The AIX operating system also supports the BSD UNIX print command (**lpr**) and the System V UNIX print command (**lp**). See the **lpr** and **lp** commands in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

See the **qprt** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Using the smit Command

You can also issue the **qprt** command with **smit**. At the prompt, type:

```
smit qprt
```

Press Enter.

Canceling a Print Job (qcan Command)

You can cancel any job in the print queue by using the **qcan** Command or the **smit** Command. When you cancel a print job, you are prompted to provide the name of the print queue where the job resides and the job number to be canceled.

This procedure applies to both local and remote print jobs.

Prerequisites

- For local print jobs, the printer must be physically attached to your system.
- For remote print jobs, your system must be configured to communicate with the remote print server.

Using the qcan Command

The **qcan** command cancels either a particular job number in a local or remote print queue, or all jobs in a local print queue. To determine the job number, type the **qchk** command.

The common format of the **qcan** command is as follows:

```
qcan -PQueueName -x JobNumber
```

For example, to cancel job number `123` on whichever printer the job is on, type:

```
qcan -x 123
```

For example, to cancel all jobs queued on printer `lp0`, type:

```
qcan -X -Plp0
```

> **Note:** The AIX operating system also supports the BSD UNIX cancel print command (**lprm**) and the System V UNIX cancel print command (**cancel**). See the **lprm** and **cancel** commands in the *AIX 5L Version 5.2 Commands Reference* for more information and the exact syntax.

Using the smit Command

To cancel a print job using SMIT, type:

```
smit qcan
```

See the **qcan** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Checking Print Job Status (qchk Command)

To display the current status information for specified job numbers, queues, printers, or users, you can use the **Web-based System Manager Fast Path**, **qchk** Command, or the **smit** Command.

Prerequisites

- For local print jobs, the printer must be physically attached to your system.
- For remote print jobs, your system must be configured to communicate with the remote print server.

Web-based System Manager Fast Path

To check the status of a print job using the Web-based System Manager fast path, type:

```
wsm printers
```

In the Printer Queues container, select the print job, then use the menus to check its status.

Using the qchk Command

You can use the **qchk** command to display the current status information regarding specified print jobs, print queues, or users.

The common format of the **qchk** command is:

```
qchk -P QueueName -# JobNumber -u OwnerName
```

See the **qchk** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

The following are examples of howto use the **qchk** command:

1. To display the default print queue, type:
   ```
   qchk -q
   ```
   Press Enter.
2. To display the long status of all queues until all queued jobs are complete, while updating the screen every 5 seconds, type:
   ```
   qchk -A -L -w 5
   ```
   To return to the command prompt, type **^C**.
3. To display the status for print queue lp0, type:
   ```
   qchk -P lp0
   ```
   Press Enter.
4. To display the status for job number 123, type:
   ```
   qchk -# 123
   ```
   Press Enter.
5. To check the status of all jobs in all queues, type:
   ```
   qchk -A
   ```

Note: The AIX operating system also supports the BSD UNIX check print queue command (**lpq**) and the System V UNIX check print queue command (**lpstat**).

See the **lpq** and **lpstat** commands in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Using the smit Command

To check a print job's status using SMIT, type:

```
smit qchk
```

Printer Status Conditions

Some of the status conditions that a print queue can have are as follows:

DEV_BUSY	Indicates that: • More than one queue is defined to a printer device (lp0), and another queue is currently using the printer device. • **qdaemon** attempted to use the printer port device (lp0), but another application is currently using that printer device. To recover from a **DEV_BUSY**, wait until the queue or application has released the printer device or cancel the job or process that is using the printer port.
DEV_WAIT	Indicates that the queue is waiting on the printer because the printer is offline, out of paper, jammed, or the cable is loose, bad, or wired incorrectly. To recover from a **DEV_WAIT**, correct the problem that caused it to wait. Sometimes, the jobs have to be removed from the queue before the problem can be corrected.
DOWN	A queue will usually go into a **DOWN** state after it has been in the **DEV_WAIT** state. This situation occurs when the printer device driver cannot tell if the printer is there due to absence of correct signallng. However, some printers might not have the capability to signal the queuing system that it is offline, and instead signals that it is off. If the printer device signals or appears to be off, the queue will go into the **DOWN** state. To recover from a **DOWN** state, correct the problem that has brought the queue down and have the system administrator bring the queue back up. The queue *must* be manually brought up before it can be used again.
HELD	Specifies that a print job is held. The print job will not be processed by the spooler until it is released.
QUEUED	Specifies that a print file is queued and is waiting in line to be printed.
READY	Specifies that everything involved with the queue is ready to queue and print a job.
RUNNING	Specifies that a print file is printing.

Prioritizing a Print Job (qpri Command)

To change the priority of a print job, use the **qpri** Command or **smit** Command. You can only assign job priority on local queues. Higher values indicate a higher priority for the print job. The default priority is 15. The maximum priority for most user print jobs is 20. However, print jobs from users with root user authority or members of the printq group (group 0) can recieve a priority of 30.

Note: You cannot assign priority to a remote print job.

Prerequisites

- For local print jobs, the printer must be physically attached to your system.
- For remote print jobs, your system must be configured to communicate with the remote print server.

Using the qpri Command(qpri Command)

The **qpri** command reassigns the priority of a print job that you submitted. If you have root user authority or belong to the printq group, you can assign priority to any job while it is in the print queue.

The basic format of the **qpri** command is:

```
qpri -# JobNumber -a PriorityLevel
```

For example, to change job number `123` to priority number `18`, type:

```
qpri -# 123 -a 18
```

For example, to prioritize a local print job as it is submitted, type:

```
qprt -PQueueName -R PriorityLevel FileName
```

See the **qpri** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Using the smit Command

To change the priority of a print job using SMIT, type:

```
smit qpri
```

Holding and Releasing a Print Job (qhld Command)

After you have sent a print job to a print queue, you can put the print job on hold by using the **Web-Based System Manager Fast Path**, the **qhld** Command, or the **smit** Command.**"Web-based System Manager Fast Path"**, the **"Using the qhld Command"**, or the **"Using the smit Command" on page 98**. You can use the same commands to later release the print job for printing.

Prerequisites

- For local print jobs, the printer must be physically attached to your system.
- For remote print jobs, your system must be configured to communicate with the remote print server.

Web-based System Manager Fast Path

To hold or release a print job using the Web-based System Manager fast path, type:

```
wsm printers
```

In the Printer Queues container, select the print job, then use the menus to put it on hold or to release it for printing.

Using the qhld Command

The **qhld** command puts a print job on hold after you have sent it. You can either put a particular print job on hold, or you can hold all the print jobs on a specified print queue. To determine the print job number, type the **qchk** command.

The common format of the **qhld** command is:

```
qhld [ -r ] {[ -#JobNumber ] [ -PQueue ] [ -uUser ]}
```

See the **qhld** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

The following are examples of how to use the **qhld** command:

1. To hold job number `452` on whichever print queue the job is on, type:

   ```
   qhld -#452
   ```

Press Enter.

2. To hold all jobs queued on print queue `hp2`, type:

   ```
   qhld -Php2
   ```

   Press Enter.
3. To release job number `452` on whichever print queue the job is on, type:

   ```
   qhld -#452 -r
   ```

   Press Enter.
4. To release all jobs queued on print queue `hp2`, type:

   ```
   qhld -Php2 -r
   ```

   Press Enter.

Using the smit Command

To hold or release a print job using SMIT, type:

```
smit qhld
```

Moving a Print Job to Another Print Queue (qmov Command)

After you have sent a print job to a print queue, you might want to move the print job to another print queue. You can move it with the **qmov** command or the **smit** command.

Prerequisites

- For local print jobs, the printer must be physically attached to your system.
- For remote print jobs, your system must be configured to communicate with the remote print server.

Using the qmov Command

You can either move a particular print job, or you can move all the print jobs on a specified print queue or all the print jobs sent by a specified user. To determine the print job number, type the **qchk** command.

The common format of the **qmov** command is:

```
qmov -mNewQueue {[ -#JobNumber ] [ -PQueue ] [ -uUser ]}
```

See the **qmov** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

The following are examples of how to use the **qmov** command:

1. To move job number `280` to print queue `hp2`, type:

   ```
   qmov -mhp2 -#280
   ```

   Press Enter.
2. To move all print jobs on print queue `hp4D` to print queue `hp2`, type:

   ```
   qmov -mhp2 -Php4D
   ```

Using the smit Command

To move a print job using SMIT, type:

```
smit qmov
```

Formatting Files for Printing (pr Command)

The **pr** command performs simple formatting of the files you sent to be printed. Pipe the output of the **pr** command to the **qprt** command to format your text.

Some useful **pr** command flags are as follows:

-d	Double-spaces the output.
-h *″String″*	Displays the specified string, enclosed in quotation marks (″ ″), instead of the file name as the page header. The flag and string should be separated by a space.
-l *Lines*	Overrides the 66-line default and resets the page length to the number of lines specified by the *Lines* variable. If the *Lines* value is smaller than the sum of both the header and trailer depths (in lines), the header and trailer are suppressed (as if the **-t** flag were in effect).
-m	Merges files. Standard output is formatted so that the **pr** command writes one line from each file specified by a *File* variable, side by side into text columns of equal fixed widths, based on the number of column positions. Do not use this flag with the **-***Column* flag.
-n [*Width*][*Character*]	Provides line numbering based on the number of digits specified by the *Width* variable. The default is 5 digits. If the *Character* (any non-digit character) variable is specified, it is appended to the line number to separate it from what follows on the line. The default character separator is the ASCII TAB character.
-o *Offset*	Indents each line by the number of character positions specified by the *Offset* variable. The total number of character positions per line is the sum of the width and offset. The default value of *Offset* is 0.
-s*Character*	Separates columns by the single character specified by the *Character* variable instead of by the appropriate number of spaces. The default value for *Character* is an ASCII TAB character.
-t	Does not display the five-line identifying header and the five-line footer. Stops after the last line of each file without spacing to the end of the page.
-w *Width*	Sets the number of column positions per line to the value specified by the *Width* variable. The default value is 72 for equal-width multicolumn output. There is no limit otherwise. If the **-w** flag is not specified and the **-s** flag is specified, the default width is 512 column positions.
-*Column*	Sets the number of columns to the value specified by the *Column* variable. The default value is 1. Do not use his option with the **-m** flag. The **-e** and **-i** flags are assumed for multicolumn output. A text column should never exceed the length of the page (see the **-l** flag). When this flag is used with the **-t** flag, use the minimum number of lines to write the output.
+*Page*	Begins the display with the page number specified by the *Page* variable. The default value is 1.

For example, to print a file named **prog.c**with headings and page numbers, type:

```
pr prog.c | qprt
```

Press Enter.

The **pr** Command , by default, adds page headings and page numbers to **prog.c** and sends it to the **qprt** command. The heading consists of the date the file was last modified, the file name, and the page number.

For example, to specify a title for a file named **prog.c** , type:

```
pr -h "MAIN PROGRAM" prog.c | qprt
```

Press Enter.

This prints **prog.c** with the title `MAIN PROGRAM` in place of the file name. The modification date and page number are still printed.

For example, to print a file named **word.lst** in multiple columns, type:

```
pr -3 word.lst | qprt
```

Press Enter.

This prints the **word.lst** file in three vertical columns.

For example, to print several files side by side on the paper:

```
pr -m -h "Members and Visitors" member.lst visitor.lst | qprt
```

This prints**member.lst** and **visitor.lst** side by side with the title `Members and Visitors`.

For example, to modify a file named `prog.c` for later use, type:

```
pr -t -e prog.c > prog.notab.c
```

Press Enter.

This replaces tab characters in `prog.c` with spaces and puts the result in `prog.notab.c`. Tab positions are at columns 9, 17, 25, 33, and so on. The **-e** flag tells the **pr** command to replace the tab characters; the **-t** flag suppresses the page headings.

For example, to print a file named `myfile` in two columns, in landscape, and in 7-point text, type:

```
pr -l66 -w172 -2 myfile | qprt -z1 -p7
```

Press Enter.

See the **pr** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Printing ASCII Files on a PostScript Printer

The Text Formatting System includes the enscript filter for converting ASCII print files to PostScript for printing on a PostScript printer. The **qprt -da** command calls this filter when a print job is submitted to a PostScript print queue.

Prerequisites

- The printer must be physically attached to your system.
- The printer must be configured and defined.
- The transcript portion of Text Formatting Services must be installed.

You might specify the following flags with the **qprt** command to customize the output when submitting ASCII files to a PostScript print queue.

-1+	Adds page headings.
-2+	Formats the output in two columns.
-3+	Prints the page headings, dates, and page numbers in a fancy style. This is sometimes referred to as **gaudy** mode.
-4+	Prints the file, even if it contains unprintable characters.
-5+	Lists characters that are not included in a font.
-h *string*	Specifies a string to be used for page headings. If this flag is not specified, the heading consists of the file name, modification date, and page number.
-l *value*	Specifies the maximum number of lines printed per page. Depending on the point size, fewer lines per page might actually appear.
-L!	Truncates lines longer than the page width.
-p	Specifies the point size. If this flag is not specified, a point size of 10 is assumed, unless two-column rotated mode (**-2+ -z1**) is specified, in which case a value of 7 is used.

-s	Specifies the font style. If this flag is not specified, the Courier font is used. Acceptable values are as follows: Courier-Oblique Helvetica Helvetica-Oblique Helvetica-Narrow Helvetica-Narrow-Oblique NewCenturySchlbk-Italic Optima Optima-Oblique Palatino-Roman Palatino-Italic Times-Roman Times-Italic **Note:** The PostScript printer must have access to the specified font.
-z1	Rotates the output 90 degrees (landscape mode).

For example, to send the ACSII file `myfile.ascii` to the PostScript printer named `Msps1`, type:

```
qprt -da -PMsps1 myfile.ascii
```

Press Enter.

For example, to send the ACSII file `myfile.ascii` to the PostScript printer named `Msps1` and print in the Helvetica font, type:

```
qprt -da -PMsps1 -sHelvetica myfile.ascii
```

Press Enter.

For example, to send the ASCII file `myfile.ascii` to the PostScript printer named `Msps1` and print in the point size 9, type:

```
qprt -da -PMsps1 -p9 myfile.ascii
```

Press Enter.

Automating the Conversion of ASCII to PostScript

Many applications that generate PostScript print files follow the convention of making the first two characters of the PostScript file `%!` which identifies the print file as a PostScript print file. To configure the system to detect ASCII print files submitted to a PostScript print queue and automatically convert them to PostScript files before sending them to the PostScript printer, perform these steps:

1. At the prompt, type:

   ```
   smit chpq
   ```

   Press Enter.
2. Type the PostScript queue name, or use the List feature to select from a list of queues.
3. Select **Printer Setup** menu option.
4. Change value of **AUTOMATIC detection of print file TYPE to be done?** field to **yes**.

Any of the following commands now convert an ASCII file to a PostScript file and print it on a PostScript printer. To convert `myfile.ascii`, type any of the following at the command line:

```
qprt -Pps myfile.ps myfile.ascii
```

```
lpr -Pps myfile.ps myfile.ascii
lp -dps myfile.ps myfile.acsii
```

where `ps` is a PostScript print queue.

Overriding Automatic Determination of Print File Types

You might need to override the automatic determination of print file type for PostScript printing in the following situations.

- To print a PostScript file named `myfile.ps` that does not begin with `%!`, type the following at the command line, for example:
  ```
  qprt -ds -Pps myfile.ps
  ```
- To print the source listing of a PostScript file named `myfile.ps` that begins with `%!`, type the following at the command line, for example:
  ```
  qprt -da -Pps myfile.ps
  ```

Command Summary for Printers, Print Jobs, and Queues

cancel	Cancels requests to a line printer
lp	Sends requests to a line printer
lpq	Examines the spool queue
lpr	Enqueues print jobs
lprm	Removes jobs from the line printer spooling queue
lpstat	Displays line printer status information
pr	Writes a file to standard output
qcan	Cancels a print job
qchk	Displays the status of a print queue
qhld	Holds or releases a print job
qmov	Moves a print job to another print queue
qpri	Prioritizes a job in the print queue
qprt	Starts a print job

Related Information

“Commands Overview” on page 26

“Processes Overview” on page 35

Chapter 5, “Input and Output Redirection” on page 45

“File Systems” on page 53

“Directory Overview” on page 56

Chapter 7, “Files” on page 67

Chapter 2, “User Environment and System Information” on page 11

Chapter 9. Backup Files and Storage Media

Once your system is in use, your next consideration should be to back up the file systems, directories, and files. All computer files are potentially easy to change or erase, either intentionally or by accident. If you use a careful and methodical approach to backing up your file systems, you should always be able to restore recent versions of files or file systems with little difficulty.

> **Note:** When a hard disk crashes, the information contained on that disk is destroyed. The only way to recover the destroyed data is to retrieve the information from your backup copy.

There are several different methods of backing up. The most frequently used method is a regular backup, which is a copy of a file system, directory, or file that is kept for file transfer or in case the original data is unintentionally changed or destroyed. Another form of backing up is the archive backup; this method is used for future reference, historical purposes, or for recovery if the original data is damaged or lost.

This chapter discusses the following:

Establishing a Backup Policy

No single backup policy can meet the needs of all users. A policy that works well for a system with one user, for example, could be inadequate for a system that serves 5 or 10 different users. Likewise, a policy developed for a system on which many files are changed daily would be inefficient for a system on which data changes infrequently. Only you can determine the best backup policy for your system, but the following general guidelines should help:

Make sure you can recover from major losses.

Can your system continue to run after any single fixed disk fails? Can you recover your system if all the fixed disks should fail? Could you recover your system if you lost your backup diskettes or tape to fire or theft? Although these things are not likely, any of them are possible. Think through each of these possible losses and design a backup policy that would enable you to recover your system after any of them.

Check your backups periodically.

Backup media and its hardware can be unreliable. A large library of backup tapes or diskettes is useless if their data cannot be read back onto a fixed disk. To make certain that your backups are usable, try to display the table of contents from the backup tape periodically (using **restore -T**, or **tar -t** for archive tapes). If you use diskettes for your backups and have more than one diskette drive, try to read diskettes from a different drive than the one on which they were created. You also might want the security of repeating each level 0 backup with a second set of diskettes. If you use a streaming tape device for backups, you can use the **tapechk** command to perform rudimentary consistency checks on the tape.

Keep old backup media.

Develop a regular cycle for reusing your backup media; however, do not reuse *all* of your backup media. Sometimes it might be months before you or another system user notices that an important file is damaged or missing. Do save old backup media for such possibilities. For example, you could have the following three cycles of backup tapes or diskettes:

- Once per week, recycle all daily diskettes except the one for Friday.
- Once per month, recycle all Friday diskettes except for the one from the last Friday of the month. This makes the last four Friday backups always available.
- Once per quarter, recycle all monthly diskettes except for the last one. Keep the last monthly diskette from each quarter indefinitely, perhaps in a different building.

Check file systems before backing them up.

A backup that was made from a damaged file system might be useless. Before making your backups, it is good policy to check the integrity of the file system with the **fsck** command.

Ensure files are not in use during a backup.

Ensure your system is not in use when you make your backups. If the system is in use, files can change while they are being backed up, and the backup copy will not be accurate.

Back up your system before major changes are made to the system.

Back up your entire system before any hardware testing or repair work is performed or before you install any new devices, programs, or other system features.

Other Factors

When planning and implementing a backup strategy, concider the following factors:

- How often does the data change? The operating system data does not change very often so you do not need to back it up frequently. User data, on the other hand, usually changes frequently and you should back it up frequently.
- How many users are on the system? The number of users affects the amount of storage media and frequency required for backups.
- How difficult would it be to re-create the data? It is important to consider that some data cannot be re-created if there is no backup available.

Having a backup strategy in place to preserve your data is very important. Evaluating the needs of your site will help you to determine the backup policy that is best for you. Perform user information backups frequently and regularly. Recovering from data loss is very difficult if a good backup strategy has not been implemented.

Backup Media

Several different types of backup media are available. The different types of backup media available to your specific system configuration depend upon both your software and hardware. The types most frequently used are the 5.25-inch diskette, 8-mm tape, 9-track tape, and the 3.5-inch diskette.

Attention: Running the **backup** command results in the loss of all material previously stored on the selected backup medium.

Diskettes

Diskettes are the standard backup medium. Unless you specify a different device using the **backup -f** command, the **backup** command automatically writes its output to the **/dev/rfd0** device, which is the diskette drive. To back up to the default tape device, type **/dev/rmt0** and press Enter.

Be careful when you handle diskettes. Because each piece of information occupies such a small area on the diskette, small scratches, dust, food, or tobacco particles can make the information unusable. Be sure to remember the following:

- Do not touch the recording surfaces.
- Keep diskettes away from magnets and magnetic field sources such as telephones, dictation equipment, and electronic calculators.
- Keep diskettes away from extreme heat and cold. The recommended temperature range is 10 degrees Celsius to 60 degrees Celsius (50 degrees Fahrenheit to 140 degrees Fahrenheit).
- Proper care helps prevent loss of information.
- Make back-up copies of your diskettes regularly.

Attention: Diskette drives and diskettes must be the correct type to store data successfully. If you use the wrong diskette in your 3.5-inch diskette drive, the data on the diskette could be destroyed.

The diskette drive uses the following 3.5-inch diskettes:

- 1 MB capacity (stores approximately 720 KB of data)
- 2 MB capacity (stores approximately 1.44 MB of data).

Tapes

Because of its high capacity and durability, tape is is often chosen for storing large files or many files, such as archive copies of file systems. It is also used for transferring many files from one system to another. Tape is not widely used for storing frequently accessed files because other media provide much faster access times.

Tape files are created using commands such as **backup**, **cpio**, and **tar**, which open a tape drive, write to it, and close it.

Formatting Diskettes (format or fdformat Command)

Attention: Formatting a diskette destroys any existing data on that diskette.

You can format diskettes in the diskette drive specified by the *Device* parameter (the **/dev/rfd0** device by default) with the **format** and **fdformat** commands. The **format** command determines the device type, which is one of the following:

- 5.25-inch low-density diskette (360 KB) containing 40x2 tracks, each with 9 sectors
- 5.25-inch high-capacity diskette (1.2 MB) containing 80x2 tracks, each with 15 sectors
- 3.5-inch low-density diskette (720 KB) containing 80x2 tracks, each with 9 sectors
- 3.5-inch high-capacity diskette (2.88 MB) containing 80x2 tracks, each with 36 sectors

The sector size is 512 bytes for all diskette types.

The **format** command formats a diskette for high density unless the *Device* parameter specifies a different density.

The **fdformat** command formats a diskette for low density unless the **-h** flag is specified. The *Device* parameter specifies the device containing the diskette to be formatted (such as the **/dev/rfd0** device for drive 0).

Before formatting a diskette, the **format** and **fdformat** commands prompt for verification. This allows you to end the operation cleanly if necessary.

For example, to format a diskette in the **/dev/rfd0** device, type:

```
format -d /dev/rfd0
```

Press Enter.

For example, to format a diskette without checking for bad tracks, type:

```
format -f
```

Press Enter.

For example, to format a 360 KB diskette in a 5.25-inch, 1.2 MB diskette drive in the **/dev/rfd1** device, type:

```
format -l -d /dev/rfd1
```

Press Enter.

For example, to force high-density formatting of a diskette when using the **fdformat** command, type:

```
fdformat -h
```

Press Enter.

See the **format** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Checking the Integrity of the File System (fsck Command)

You can check and interactively repair inconsistent file systems with the **fsck** command. It is important run this command on every file system as part of system initialization. You must be able to read the device file on which the file system resides (for example, the **/dev/hd0** device). Normally, the file system is consistent, and the **fsck** command merely reports on the number of files, used blocks, and free blocks in the file system. If the file system is inconsistent, the **fsck** command displays information about the inconsistencies found and prompts you for permission to repair them. The **fsck** command is conservative in its repair efforts and tries to avoid actions that might result in the loss of valid data. In certain cases, however, the **fsck** command recommends the destruction of a damaged file.

> **Attention:** Always run the **fsck** command on file systems after a system malfunction. Corrective actions can result in some loss of data. The default action for each consistency correction is to wait for the operator to enter `yes` or `no`. If you do not have write permission for an affected file, the **fsck** command will default to a `no` response.

For example, to check all the default file systems, type:

```
fsck
```

Press Enter.

This form of the **fsck** command asks you for permission before making any changes to a file system.

For example, to fix minor problems automatically with the default file systems , type:

```
fsck -p
```

Press Enter.

For example, to check the **/dev/hd1**file system , type:

```
fsck /dev/hd1
```

Press Enter.

This checks the unmounted file system located on the `/dev/hd1` device.

> **Note:** The **fsck** command does not make corrections to a mounted file system.

See the **fsck** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Copying to or from Diskettes (flcopy Command)

You can copy a diskette (opened as **/dev/rfd0**) to a file named `floppy` created in the current directory with the **flcopy** command. The message: `Change floppy, hit return when done` displays as needed. The **flcopy** command then copies the `floppy` file to the diskette.

For example, to copy `/dev/rfd1` to the `floppy` file in the current directory, type:

```
flcopy -f /dev/rfd1 -r
```

Press Enter.

For example, to copy the first 100 tracks of the diskette, type:

```
flcopy -f /dev/rfd1 -t 100
```

Press Enter.

See the **flcopy** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Copying Files to Tape or Disk (cpio -o Command)

You can use the **cpio -o** Command to read file path names from standard input and copy these files to standard output, along with path names and status information.Path names cannot exceed 128 characters. Avoid giving the **cpio** command path names made up of many uniquely linked files, as it might not have enough memory to keep track of the path names and would lose linking information.

For example, to copy files in the current directory whose names end with `.c` onto diskette, type:

```
ls *.c | cpio -ov >/dev/rfd0
```

Press Enter. The **-v** flag displays the names of each file.

For example, to copy the current directory and all subdirectories onto diskette, type:

```
find . -print | cpio -ov >/dev/rfd0
```

Press Enter.

This saves the directory tree that starts with the current directory (`.`) and includes all of its subdirectories and files. To use a shorter command string, type:

```
find . -cpio /dev/rfd0 -print
```

Press Enter.

The `-print` entry displays the name of each file as it is copied.

See the **cpio** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Copying Files from Tape or Disk (cpio -i Command)

The **cpio -i** command reads from standard input an archive file created by the **cpio -o** command and copies from it the files with names that match the *Pattern* parameter. These files are copied into the current directory tree. You can list more than one *Pattern* parameter, using the file name notation described in the **ksh** command. The default for the *Pattern* parameter is an asterisk (*), selecting all files in the current directory. In an expression such as `[a-z]`, the hyphen (-) means *through* according to the current collating sequence.

> **Note:** The patterns ″`*.c`″ and ″`*.o`″ must be enclosed in quotation marks to prevent the shell from treating the asterisk (*) as a pattern-matching character. This is a special case in which the **cpio** command itself decodes the pattern-matching characters.

For example, to list the files that have been saved onto a diskette with the **cpio** command, type:

```
cpio -itv </dev/rfd0
```

Press Enter.

This displays the table of contents of the data previously saved onto the `/dev/rfd0` file in the **cpio** command format. The listing is similar to the long directory listing produced by the **ls -l** command. To list only the file path names, use only the **-it** flags.

For example, to copy the files previously saved with the **cpio** command from a diskette, type:

```
cpio -idmv </dev/rfd0
```

Press Enter.

This copies the files previously saved onto the `/dev/rfd0` file by the **cpio** command back into the file system (specify the **-i** flag). The **-d** flag allows the **cpio** command to create the appropriate directories if a directory tree is saved. The **-m** flag maintains the last modification time in effect when the files are saved. The **-v** flag causes the **cpio** command to display the name of each file as it is copied.

For example, to copy selected files from diskette, type:

```
cpio -i "*.c" "*.o" </dev/rfd0
```

Press Enter.

This copies the files that end with `.c` or `.o` from diskette.

See the **cpio** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Copying to or from Tapes (tcopy Command)

You can use the **tcopy** command to copy magnetic tapes.

For example, to copy from one streaming tape to a 9-track tape, type:

```
tcopy /dev/rmt0 /dev/rmt8
```

Press Enter.

See the **tcopy** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Checking the Integrity of a Tape (tapechk Command)

You can perform rudimentary consistency checking on an attached streaming tape device with the **tapechk** command. Some hardware malfunctions of a streaming tape drive can be detected by simply reading a tape. The **tapechk** command provides a way to perform tape reads at the file level.

For example, to check the first three files on a streaming tape device, type:

```
tapechk 3
```

Press Enter.

See the **tapechk** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Compressing Files (compress and pack Commands)

You can compress files for storage with the **compress** Command and **pack** Command, and use the **uncompress** and **unpack** to expand the restored files. The process of compressing and expanding files takes time but, after the files are packed, the data uses less space on the backup medium.

To compress a file system, use one of the following methods:

- Use the **-p** option with the **backup** command
- Use the **compress** or **pack** commands

The reasons for compressing files generally fall into the following categories:

- Saving storage and archiving system resources:
  - Compress file systems before doing backups to preserve tape space.
  - Compress log files created by shell scripts that run at night; it is easy to have the script compress the file before it exits.
  - Compress files that are not currently being accessed. For example, the files belonging to a user who is away for extended leave can be compressed and placed into a **tar** archive on disk or to a tape and later be restored.
- Saving money and time by compressing files before sending them over a network.

Notes:

1. The **compress** command might run out of working space in the file system while compressing. The command creates the compressed files before it deletes any of the uncompressed files so it needs a space about 50% larger than the total size of the files.
2. A file might fail to compress because it is already compressed. If the **compress** command cannot reduce file sizes, the command fails.

Using the compress Command

The **compress** command reduces the size of files using adaptive Lempel-Zev coding. Each original file specified by the *File* parameter is replaced by a compressed file with a **.Z** appended to its name. The compressed file retains the same ownership, modes, and access and modification times of the original file. If no files are specified, the standard input is compressed to the standard output. If compression does not reduce the size of a file, a message is written to standard error and the original file is not replaced.

To restore compressed files to their original form, use the **uncompress** command.

The amount of compression depends on the size of the input, the number of bits per code specified by the *Bits* variable, and the distribution of common substrings. Typically, source code or English text is reduced by 50 to 60 percent. The compression of the **compress** command is generally more compact and takes less time to compute than the compression achieved by the **pack** command, which uses adaptive Huffman coding.

For example, to compress the **foo** file and write the percentage compression to standard error, type:

```
compress -v foo
```

Press Enter.

See the **compress** command in the*AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Using the pack Command

The **pack** command stores the file or files specified by the *File* parameter in a compressed form using Huffman coding. The input file is replaced by a packed file with a name derived from the original file name (*File***.z**), with the same access modes, access and modification dates, and owner as the original file. The input file name can contain no more than 253 bytes to allow space for the added **.z** suffix. If the **pack** command is successful, the original file is removed. To restore packed files to their original form, use the **unpack** command.

If the **pack** command cannot create a smaller file, it stops processing and reports that it is unable to save space. (A failure to save space generally happens with small files or files with uniform character distribution.) The amount of space saved depends on the size of the input file and the character frequency distribution. Because a decoding tree forms the first part of each **.z** file, you do not save space with files smaller than three blocks. Typically, text files are reduced 25 to 40 percent.

The exit value of the **pack** command is the number of files that it could not pack. Packing is not done under any of the following conditions:

- The file is already packed.
- The input file name has more than 253 bytes.
- The file has links.
- The file is a directory.
- The file cannot be opened.
- No storage blocks are saved by packing.
- A file called *File***.z** already exists.
- The **.z** file cannot be created.
- An I/O error occurred during processing.

For example, to compress the files `chap1` and `chap2`, type:

```
pack chap1 chap2
```

Press Enter.

This compresses `chap1` and `chap2`, replacing them with files named **chap1.z** and **chap2.z**. The **pack** command displays the percent decrease in size for each file.

See the **pack** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Expanding Compressed Files (uncompress and unpack Commands)

You can expand compressed files with the **uncompress** and **unpack** commands.

Using the uncompress Command

The **uncompress** command restores original files that were compressed by the **compress** command. Each compressed file specified by the *File* variable is removed and replaced by an expanded copy. The expanded file has the same name as the compressed version, but without the **.Z** extension. The expanded file retains the same ownership, modes, and access and modification times as the original file. If no files are specified, standard input is expanded to standard output.

Although similar to the **uncompress** command, the **zcat** command always writes the expanded output to standard output.

For example, to uncompress the `foo` file, type:

```
uncompress foo
```

Press Enter.

See the **uncompress** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Using the unpack Command

The **unpack** command expands files created by the **pack** command. For each file specified, the **unpack** command searches for a file called *File***.z**. If this file is a packed file, the **unpack** command replaces it by its expanded version. The **unpack** command renames the new file by removing the **.z** suffix from *File*. The new file has the same access modes, access and modification dates, and owner as the original packed file.

The **unpack** command operates only on files ending in **.z**. As a result, when you specify a file name that does not end in **.z**, the **unpack** command adds the suffix and searches the directory for a file name with that suffix.

The exit value is the number of files that the **unpack** command was unable to unpack. A file cannot be unpacked if any of the following situations exits:

- The file name (exclusive of **.z**) has more than 253 bytes.
- The file cannot be opened.
- The file is not a packed file.
- A file with the unpacked file name already exists.
- The unpacked file cannot be created.

> **Note:** The **unpack** command writes a warning to standard error if the file it is unpacking has links. The new unpacked file has a different i-node (index node) number than the packed file from which it was created. However, any other files linked to the original i-node number of the packed file still exist and are still packed.

For example, to unpack the packed files `chap1.z` and `chap2`, type:

```
unpack chap1.z chap2
```

Press Enter.

This expands the packed files `chap1.z` and `chap2.z`, and replaces them with files named `chap1` and `chap2`. Note that you can provide the **unpack** command with file names with or without the **.z** suffix.

See the **unpack** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Backing Up Files (backup Command)

> **Attention:** If you attempt to back up a mounted file system, a message displays. The **backup** command continues, but inconsistencies in the file system can occur. This situation does not apply to the root (**/**) file system.

You can create copies of your files on backup media, such as a magnetic tape or diskette, with the **backup** Command or **smit** Command. The copies are in one of the following backup formats:

- Specific files backed up by name, using the **-i** flag.
- Entire file system backed up by i-node number, using the *-Level* and *FileSystem* parameters.

Notes:

1. The possibility of data corruption always exists when a file is modified during system backup. Therefore, make sure that system activity is at a minimum during the system backup procedure.
2. If a backup is made to 8-mm tape with the device block size set to 0 (zero), it is not possible to directly restore from the tape. If you have done backups with the 0 setting, you can restore from them by using special procedures described under the **restore** command.

> **Attention:** Be sure the flags you specify match the backup media.

Using the backup Command

For example, to back up selected files in your **$HOME** directory by name, type:

```
find $HOME -print | backup -i -v
```

Press Enter.

The **-i** flag prompts the system to read from standard input the names of files to be backed up. The **find** command generates a list of files in the user's directory. This list is piped to the **backup** command as standard input. The **-v** flag displays a progress report as each file is copied. The files are backed up on the default backup device for the local system.

For example, to back up the root file system, type:

```
backup -0 -u /
```

Press Enter.

The 0 level and the **/** tell the system to back up the **/** (root) file system. The file system is backed up to the **/dev/rfd0** file. The **-u** flag tells the system to update the current backup level record in the **/etc/dumpdates** file.

For example, to back up all files in the **/** (root) file system that were modified since the last 0 level backup, type:

```
backup -1 -u /
```

Press Enter.

See the **backup** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Using the smit Command

You can also use **smit** to run the **backup** command.

1. At the prompt, type:
   ```
   smit backup
   ```
   Press Enter.
2. Type the path name of the directory on which the file system is normally mounted in the **DIRECTORY full pathname** field:
   ```
   /home/bill
   ```
   Press Enter.
3. In the **BACKUP** device or **FILE** fields, type the output device name, as in the following example for a raw magnetic tape device:
   ```
   /dev/rmt0
   ```
   Press Enter.
4. Use the Tab key to toggle the optional **REPORT each phase of the backup** field if you want error messages printed to the screen.
5. In a system management environment, use the default for the **MAX number of blocks to write on backup medium** field, because this field does not apply to tape backups.
6. Press Enter to back up the named directory or file system.
7. Run the **restore -t** command. If this command generates an error message, you must repeat the entire backup.

Restoring Backed-Up Files (restore Command)

You can read files written by the**backup** command from backup media and restore them on your local system with the **restore** command or **smit** command.

Notes:

1. Files must be restored using the same method by which they were backed up. For example, if a file system was backed up by name, it must be restored by name.
2. When more than one diskette is required, the **restore** command reads the diskette that is mounted, prompts you for a new one, and waits for your response. After inserting the new diskette, press the Enter key to continue restoring files.

Using the restore Command

For example, to list the names of files previously backed up, type:

```
restore -T
```

Press Enter.

Information is read from the **/dev/rfd0** default backup device. If individual files are backed up, only the file names are displayed. If an entire file system is backed up, the i-node number is also shown.

For example, to restore files to the main file system, type:

```
restore -x -v
```

Press Enter.

The **-x** flag extracts all the files from the backup media and restores them to their proper places in the file system. The **-v** flag displays a progress report as each file is restored. If a file system backup is being restored, the files are named with their i-node numbers. Otherwise, only the names are displayed.

For example, to copy the **/home/mike/manual/chap1**file , type:

```
restore -xv /home/mike/manual/chap1
```

Press Enter.

This command extracts the **/home/mike/manual/chap1** file from the backup medium and restores it. The **/home/mike/manual/chap1** file must be a name that the **restore -T** command can display.

For example, to copy all the files in a directory named **manual**, type:

```
restore -xdv manual
```

Press Enter.

This command restores the **manual** directory and the files in it. If the directory does not exist, a directory named **manual** is created in the current directory to hold the files being restored.

See the **restore** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Using the smit Command

You can also use **smit** to run the **restore** command.

1. At the prompt, type:

   ```
   smit restore
   ```

   Press Enter.
2. Make your entry in the **Target DIRECTORY** field. This is the directory where you want the restored files to reside.
3. Proceed to the **BACKUP device** or **FILE** field and type the output device name, and press Enter, as in the following example for a raw magnetic tape device:

   ```
   /dev/rmt0
   ```

   If the device is not available, a message similar to the following is displayed:

   ```
   Cannot open /dev/rmtX, no such file or directory.
   ```

   This message indicates that the system cannot reach the device driver because there is no file for `rmtX` in the `/dev` directory. Only items in the `available` state are in the`/dev`directory.
4. For the **NUMBER of blocks to read in a single input** field, the default is recommended.
5. Press Enter to restore the specified file system or directory.

Archiving Files (tar Command)

The archive backup is another form of backing you can use; this method is used for a copy of one or more files, or an entire database that is saved for future reference, historical purposes, or for recovery if the original data is damaged or lost. Usually an archive is used when that specific data is removed from the system.

You can write files to or retrieve files from an archive storage with the **tar** command. The **tar** command looks for archives on the default device (usually tape), unless you specify another device.

When writing to an archive, the **tar** command uses a temporary file (the **/tmp/tar*** file) and maintains in memory a table of files with several links. You receive an error message if the **tar** command cannot create the temporary file or if there is not enough memory available to hold the link tables.

For example, to write the `file1` and `file2` files to a new archive on the default tape drive, type:

```
tar -c file1 file2
```

Press Enter.

For example, to extract all files in the `/tmp` directory from the archive file on the `/dev/rmt2` tape device and use the time of extraction as the modification time, type:

```
tar -xm -f/dev/rmt2 /tmp
```

Press Enter.

For example, to display the names of the files in the `out.tar` disk archive file from the current directory, type:

```
tar -vtf out.tar
```

Press Enter.

See the **tar** command in the *AIX 5L Version 5.2 Commands Reference* for more information and the exact syntax.

Command Summary for Backup Files and Storage Media

backup	Backs up files and file systems
compress	Compresses and expands data
cpio	Copies files into and out of archive storage and directories
fdformat	Formats diskettes
flcopy	Copies to and from diskettes
format	Formats diskettes
fsck	Checks file system consistency and interactively repairs the file system
pack	Compresses files
restore	Copies previously backed-up file systems or files, created by the **backup** command, from a local device
tapechk	Checks consistency of the streaming tape device
tar	Manipulates archives
tcopy	Copies a magnetic tape
uncompress	Compresses and expands data
unpack	Expands files

Related Information

"Commands Overview" on page 26

"Processes Overview" on page 35

Chapter 5, "Input and Output Redirection" on page 45

"File Systems" on page 53

“Directory Overview” on page 56

Chapter 7, “Files” on page 67

Chapter 10, “File and System Security” on page 117

Chapter 10. File and System Security

The goal of computer security is the protection of information stored on the computer system, a valuable resource. Information security is aimed at the following:

Integrity	The value of all information depends upon its accuracy. If unauthorized changes are made to data, this data loses some or all of its value.
Privacy	The value of much information depends upon its secrecy.
Availability	Information must be readily available.

It is helpful to plan and implement your security policies before you begin using the system. Security policies are very time-consuming to change later, so upfront planning can save a lot of time later.

This chapter discusses the following:

Security Threats

Threats to information security can arise from the following types of behavior:

Carelessness	Information security is often violated due to the carelessness of the authorized users of the system. If you are careless with your password, for instance, no other security mechanisms can prevent unauthorized access to your account and data.
Browsing	Many security problems are caused by browsers, authorized users of the system exploring the system looking for carelessly protected data.
Penetration	Penetration represents deliberate attacks upon the system. An individual trying to penetrate the system will study it for security vulnerabilities and deliberately plan attacks designed to exploit those weaknesses.

Although system penetration usually represents the greatest threat to information security, do not underestimate problems caused by carelessness or browsing.

Basic Security

Every system should maintain the level of security represented by the following basic security policies:

Backups

Physically secure, reliable, and up-to-date system backups are the single most important security policy. With a good system backup, you can recover from any system problems with minimal loss. Document your backup policy and include information regarding the following:

- How often backups will be made
- What types of backups (system, data, or incremental) will be made
- How backup tapes will be verified
- How backup tapes will be stored

For more information, see ″Chapter 9, “Backup Files and Storage Media” on page 103″.

Identification and Authentication

Identification and authentication establish your identity. You are required to log in to the system. You supply your user name and a password, if the account has one (in a secure system, all accounts should either have passwords or be invalidated). If the password is correct, you are logged in to that account; you acquire the access rights and privileges of the account.

Because the password is the only protection for your account, select and guard your password carefully. Many attempts to break into a system start with attempts to guess passwords. The operating system provides significant password protection by storing user passwords separately from other user information. The encrypted passwords and other security-relevant data for users are stored in the **/etc/security/passwd** file. This file should be accessible only by the root user. With this restricted access to the encrypted passwords, an attacker cannot decipher the password with a program that simply cycles through all possible or likely passwords.

It is still possible to guess passwords by repeatedly attempting to log in to an account. If the password is trivial or is infrequently changed, such attempts might easily succeed.

Login User IDs

The operating system also identifies users by their login user ID. The login user ID allows the system to trace all user actions to their source. After a user logs in to the system but before the initial user program is run, the system sets the login ID of the process to the user ID found in the user database. All subsequent processes during the login session are tagged with this ID. These tags provide a trail of all activities performed by the login user ID.

The user can reset the effective user ID, real user ID, effective group ID, real group ID, and supplementary group ID during the session, but cannot change the login user ID.

Unattended Terminals

All systems are vulnerable if terminals are left logged in and unattended. The most serious problem occurs when a system manager leaves a terminal unattended that has been enabled with root authority. In general, users should log out anytime they leave their terminals.

You can force a terminal to log out after a period of inactivity by setting the **TMOUT** and **TIMEOUT** parameters in the **/etc/profile** file. The **TMOUT** parameter works in the **ksh** (Korn) shell, and the **TIMEOUT** parameter works in the **bsh** (Bourne) shell. For more information about the **TMOUT** parameter, see “Parameter Substitution in the Korn Shell or POSIX Shell” on page 152. For more information about the **TIMEOUT** parameter, see “Variable Substitution in the Bourne Shell” on page 193.

The following example, taken from a **.profile** file, forces the terminal to log out after an hour of inactivity:

```
TO=3600
echo "Setting Autologout to $TO"
TIMEOUT=$TO
TMOUT=$TO
export TIMEOUT TMOUT
```

Note: Users can override the **TMOUT** and **TIMEOUT** values in the **/etc/profile** file by specifying different values in the **.profile** file in your home directory.

File Ownership and User Groups

Initially, a file's owner is identified by the user ID of the person who created the file. The owner of a file determines who may read, write (modify), or execute the file. Ownership can be changed with the **chown** command.

Every user ID is assigned to a group with a unique group ID. The system manager creates the groups of users when setting up the system. When a new file is created, the operating system assigns permissions to the user ID that created it, to the group ID containing the file owner, and to a group called `others`, consisting of all other users. The **id** command shows your user ID (UID), group ID (GID), and the names of all groups you belong to.

In file listings (such as the listings shown by the **ls** command), the groups of users are always represented in the following order: user, group, and others. If you need to find out your group name, the **groups** command shows all the groups for a user ID.

Changing File or Directory Ownership (chown Command)

To change the owner of your files, use the **chown** command.

When the **-R** option is specified, the **chown** command recursively descends through the directory structure from the specified directory. When symbolic links are encountered, the ownership of the file or directory pointed to by the link is changed; the ownership of the symbolic link is not changed.

Note: Only the root user can change the owner of another file. Errors are not displayed when the **-f** option is specified.

For example, to change the owner of the **program.c** file, type:

```
chown jim program.c
```

Press Enter.

The user-access permissions for the **program.c** file now apply to **jim**. As the owner, **jim** can use the **chmod** command to permit or deny other users access to the **program.c** file.

See the **chown** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

File and Directory Access Modes

Every file has an owner. For new files, the user who creates the file is the owner of that file. The owner assigns an access mode to the file. Access modes grant other system users permission to read, modify, or execute the file. Only the file's owner or users with root authority can change the access mode of a file.

There are the three classes of users: user/owner, group, and all others. Access is granted to these user classes in some combination of three modes: read, write, or execute. When a new file is created, the default permissions are read, write, and execute permission for the user who created the file. The other two groups have read and execute permission. The following table illustrates the default file-access modes for the three classes of user groups:

Classes	**Read**	**Write**	**Execute**
Owner	Yes	Yes	Yes

Group	Yes	No	Yes
Others	Yes	No	Yes

The system determines who has permission and the level of permission they have for each of these activities. Access modes are represented both symbolically and numerically in the operating system.

Symbolic Representation of Access Modes

Access modes are represented symbolically, as follows:

r	Indicates read permission, which allows users to view the contents of a file.
w	Indicates write permission, which allows users to modify the contents of a file.
x	Indicates execute permission. For executable files (ordinary files that contain programs), execute permission means that the program can be run. For directories, execute permission means the contents of the directory can be searched.

The access modes for files or directories are represented by nine charactors. The first three charactors represent the current **Owner** permissions, the second sent of three charactors represents the current **Group** permissions, and the third set of three charactors represents the current settings for the **Other** permissions. A Hyphen (**-**) in the nine charactor set indicates that no permission is given. For example, a file with the access modes set to `rwxr-xr-x` gives read and execute permission to all three groups, but write permission only to the owner of the file. This is the symbolic representation of the default setting.

The **ls** command, when used with the **-l** (lower case L) flag, gives a detailed listing of the current directory. The first 10 characters in the **ls -l** listing show the file type and permissions for each of the three groups. The **ls -l** command also tells you the owner and group associated with each file and directory.

The first character indicates the type of file. The remaining nine characters contain the file permission information for each of the three classes of users. The following symbols are used to represent the type of file:

-	Regular files
d	Directory
b	Block special files
c	Character special files
p	Pipe special files
l	Symbolic links
s	Sockets.

For example, this is a sample **ls -l** listing:

```
-rwxrwxr-x   2   janet   acct   512 Mar 01 13:33 january
```

Here, the first hyphen (-) indicates a regular file. The next nine charactors (`rwxrwxr-x` represent the User, Group, and Other access modes, as discussed above. `janet` is the file owner and `acct` is the name of Janet's group. `512` is the file size in bytes, `Mar 01 13:33` is the last date and time of modification, and `january` is the file name. The `2` indicates how many links exist to the file.

Numeric Representation of Access Modes

Numerically, read access is represented by a value of 4, write permission is represented by a value of 2, and execute permission is represented by a value of 1. The total value between 1 and 7 represents the access mode for each group (user, group, and other). The following table illustrates the numeric values for each level of access:

Total Value	Read	Write	Execute
0	-	-	-
1	-	-	1
2	-	2	-
3	-	2	1
4	4	-	-
5	4	-	1
6	4	2	-
7	4	2	1

When a file is created, the default file access mode is 755. This means the user has read, write, and execute permissions (4+2+1=7), the group has read and execute permission (4+1=5), and all others have read and execute permission (4+1=5). To change access permission modes for files you own, run the **chmod** (change mode) command.

Displaying Group Information (lsgroup Command)

To display the attributes of all the groups on the system (or of specified groups), use the **lsgroup** command. If one or more attributes cannot be read, the **lsgroup** command lists as much information as possible. The attribute information displays as *Attribute***=***Value* definitions, each separated by a blank space.

Listing All of the Groups on the System

To list all of the groups on the system, type:

```
lsgroup ALL
```

Press Enter.

The system displays each group, group ID, and all of the users in the group in a list similar to the following:

```
system  0       arne,pubs,ctw,geo,root,chucka,noer,su,dea,
backup,build,janice,denise
staff   1       john,ryan,flynn,daveb,jzitt,glover,maple,ken
gordon,mbrady
bin     2       root,bin
sys     3       root,su,bin,sys
```

Displaying Specific Attributes for All Groups

To display specific attributes for all groups, do either of the following:

- You can list attributes in the form `Attribute=Value` separated by a blank space. This is the default style. For example, to list the ID and users for all of the groups on the system, type:

  ```
  lsgroup -a id users ALL | pg
  ```

  Press Enter. The addition of the lists the attributes.

  A list similar to the following displays:

  ```
  system id=0 users=arne,pubs,ctw,geo,root,chucka,noer,su,dea,backup,build
  staff id=1 users=john,ryan,flynn,daveb,jzitt,glover,maple,ken
  ```

- You can also list the information in stanza format. For example, to list the ID and users for all of the groups on the system in stanza format, type:

  ```
  lsgroup -a -f id users ALL | pg
  ```

Press Enter.

A list similar to the following displays:

```
system:
   id=0
   users=pubs,ctw,geo,root,chucka,noer,su,dea,backup,build

staff:
   id=1
   users=john,ryan,flynn,daveb,jzitt,glover,maple,ken

bin:
   id=2
   users=root,bin

sys:
   id=3
   users=root,su,bin,sys
```

Displaying All Attributes for a Specific Group

To display all attributes for a specific group, you can use one of two styles for listing specific attributes for all groups:

- You can list each attribute in the form `Attribute=Value` separated by a blank space. This is the default style. For example, to list all attributes for the group system, type:

  ```
  lsgroup system
  ```

  Press Enter.

  A list similar to the following displays:

  ```
  system id=0 users=arne,pubs,ctw,geo,root,chucka,noer,su,dea,backup,build,janice,denise
  ```

- You can also list the information in stanza format. For example, to list all attributes for the group `bin` in stanza format, type:

  ```
  lsgroup -f system
  ```

  Press Enter.

  A list similar to the following displays:

  ```
  system:
     id=0   users=arne,pubs,ctw,geo,root,chucka,noer,su,dea,
  backup,build,janice,denise
  ```

Listing Specific Attributes for a Specific Group

To list specific attributes for a specific group, type:

```
lsgroup -a Attributes Group
```

Press Enter.

For example, to list the ID and users for group `bin`, type:

```
lsgroup -a id users bin
```

Press Enter.

A list similar to the following displays:

```
bin id=2 users=root,bin
```

See the **lsgroup** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Changing File or Directory Permissions (chmod Command)

To modify the read, write, and execute permissions of specified files and modify the search permission modes of specified directories, use the **chmod** command.

- For example, to add a type of permission to the **chap1** and **chap2**files, type:

  ```
  chmod g+w chap1 chap2
  ```

  Press Enter.

  This adds write permission for group members to the files `chap1` and `chap2`**.**
- For example, to make several permission changes at once to the `mydir` directory, type:

  ```
  chmod go-w+x mydir
  ```

  Press Enter.

  This denies (**-**) group members (**g**) and others (**o**) the permission to create or delete files (**w**) in the **mydir** directory and allows (**+**) group members and others to search the **mydir** directory or use (**x**) it in a path name. This is equivalent to the following command sequence:

  ```
  chmod g-w mydir
  chmod o-w mydir
  chmod g+x mydir
  chmod o+x mydir
  ```
- For example, to permit only the owner to use a shell procedure named `cmd` as a command, type:

  ```
  chmod u=rwx,go= cmd
  ```

  Press Enter.

  This gives read, write, and execute permission to the user who owns the file (**u=rwx**). It also denies the group and others the permission to access `cmd` in any way (**go=**).
- For example, to use the numeric mode form of the **chmod** command to change the permissions of the `text`, file type:

  ```
  chmod 644 text
  ```

  Press Enter.

  This sets read and write permission for the owner, and it sets read-only mode for the group and others.

See the **chmod** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Access Control Lists

Access control consists of protected information resources that specify who can be granted access to such resources. The operating system allows for need-to-know or discretionary security. The owner of an information resource can grant other users read or write access rights for that resource. A user who is granted access rights to a resource can transfer those rights to other users. This security allows for user-controlled information flow in the system; the owner of an information resource defines the access permissions to the object.

Users have user-based access only to the objects that they own. Typically, users receive either the group permissions or the default permissions for a resource. The major task in administering access control is to define the group memberships of users, because these memberships determine the users' access rights to the files that they do not own.

Access control lists (ACLs) increase the quality of file access controls by adding extended permissions that modify the base permissions assigned to individuals and groups. With extended permissions, you can permit or deny file access to specific individuals or groups without changing the base permissions.

> **Note:** The access control list for a file cannot exceed one memory page (approximately 4096 bytes) in size.

To maintain access control lists, use the **aclget**, **acledit**, and the **aclput** commands.

The **chmod** command in numeric mode (with octal notations) can set base permissions and attributes. The **chmod** subroutine, which the command calls, disables extended permissions. If you use the numeric mode of the **chmod** command on a file that has an ACL, extended permissions are disabled. The symbolic mode of the **chmod** command does not disable extended permissions. For information on numeric and symbolic mode, refer to the **chmod** command.

Base Permissions

Base permissions are the traditional file-access modes assigned to the file owner, file group, and other users. The access modes are: read (r), write (w), and execute/search (x).

In an access control list, base permissions are in the following format, with the *Mode* parameter expressed as rwx (with a hyphen (-) replacing each unspecified permission):

```
base permissions:
   owner(name): Mode
   group(group): Mode
   others: Mode
```

Attributes

Three attributes can be added to an access control list:

setuid (SUID)
: Set-user-ID mode bit. This attribute sets the effective and saved user IDs of the process to the owner ID of the file on execution.

setgid (SGID)
: Set-group-ID mode bit. This attribute sets the effective and saved group IDs of the process to the group ID of the file on execution.

savetext (SVTX)
: Saves the text in a text file format.

These attributes are added in the following format:

```
attributes: SUID, SGID, SVTX
```

Extended Permissions

Extended permissions allow the owner of a file to define access to that file more precisely. Extended permissions modify the base file permissions (owner, group, others) by permitting, denying, or specifying access modes for specific individuals, groups, or user and group combinations. Permissions are modified through the use of keywords.

The **permit**, **deny**, and **specify** keywords are defined as follows:

permit	Grants the user or group the specified access to the file
deny	Restricts the user or group from using the specified access to the file
specify	Precisely defines the file access for the user or group

If a user is denied a particular access by either a **deny** or a **specify** keyword, no other entry can override that access denial.

The **enabled** keyword must be specified in the ACL for the extended permissions to take effect. The default value is the **disabled** keyword.

In an ACL, extended permissions are in the following format:

```
extended permissions:
  enabled | disabled
    permit   Mode  UserInfo...:
    deny     Mode  UserInfo...:
    specify  Mode  UserInfo...:
```

Use a separate line for each **permit**, **deny**, or **specify** entry. The *Mode* parameter is expressed as **rwx** (with a hyphen (-) replacing each unspecified permission). The *UserInfo* parameter is expressed as `u:UserName`, or `g:GroupName`, or a comma-separated combination of `u:UserName` and `g:GroupName`.

> **Note:** If more than one user name is specified in an entry, that entry cannot be used in an access control decision, because a process has only one user ID.

Access Control List Example

The following is an example of an ACL:

```
attributes: SUID
base permissions:
      owner(frank):  rw-
      group(system): r-x
      others: ---
extended permissions:
      enabled
        permit  rw-  u:dhs
        deny    r--  u:chas, g:system
        specify r--  u:john, g:gateway, g:mail
        permit  rw-  g:account, g:finance
```

The parts of the ACL and their meanings are the following:

- The first line indicates that the **setuid** bit is turned on.
- The next line, which introduces the base permissions, is optional.
- The next three lines specify the base permissions. The owner and group names in parentheses are for information only. Changing these names does not alter the file owner or file group. Only the **chown** command and the **chgrp** command can change these file attributes.
- The next line, which introduces the extended permissions, is optional.
- The next line indicates that the extended permissions that follow are enabled.
- The last four lines are the extended entries. The first extended entry grants user `dhs` read (r) and write (w) permission on the file.
- The second extended entry denies read (r) access to user `chas` only when he is a member of the `system` group.
- The third extended entry specifies that as long as user `john` is a member of both the `gateway` group and the `mail` group, has read (r) access. If user `john` is not a member of both groups, this extended permission does not apply.
- The last extended entry grants any user in *both* the `account` group and the `finance` group read (r) and write (w) permission.

> **Note:** More than one extended entry can be applied to a process, with restrictive modes taking precedence over permissive modes.

See the **acledit** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Access Authorization

The owner of the information resource is responsible for managing access rights. Resources are protected by permission bits, which are included in the mode of the object. The permission bits define the access permissions granted to the owner of the object, the group of the object, and for the `others` default class. The operating system supports three different modes of access (read, write, and execute) that can be granted separately.

When a user logs in to an account (using the **login** or **su** commands), the user IDs and group IDs assigned to that account are associated with the user's processes. These IDs determine the access rights of the process.

For files, directories, named pipes, and devices (special files), access is authorized as follows:

- For each access control entry (ACE) in the access control list (ACL), the identifier list is compared to the identifiers of the process. If there is a match, the process receives the permissions and restrictions defined for that entry. The logical unions for both permissions and restrictions are computed for each matching entry in the ACL. If the requesting process does not match any of the entries in the ACL, it receives the permissions and restrictions of the default entry.
- If the requested access mode is permitted (included in the union of the permissions) and is not restricted (included in the union of the restrictions), access is granted. Otherwise, access is denied.

A process with a user ID of 0 is known as a *root user process*. These processes are generally allowed all access permissions. But if a root user process requests execute permission for a program, access is granted only if execute permission is granted to at least one user.

The identifier list of an ACL matches a process if all identifiers in the list match the corresponding type of effective identifier for the requesting process. A USER-type identifier matched is equal to the effective user ID of the process, and a GROUP-type identifier matches if it is equal to the effective group ID of the process or to one of the supplementary group IDs. For instance, an ACE with an identifier list such as the following:

```
USER:fred, GROUP:philosophers, GROUP:software_programmer
```

would match a process with an effective user ID of `fred` and a group set of:

```
philosophers, philanthropists, software_programmer, doc_design
```

but would not match for a process with an effective user ID of `fred` and a group set of:

```
philosophers, iconoclasts, hardware_developer, graphic_design
```

Note that an ACE with an identifier list of the following would match for both processes:

```
USER:fred, GROUP:philosophers
```

In other words, the identifier list in the ACE functions is a set of conditions that must hold for the specified access to be granted.

All access permission checks for these objects are made at the system call level when the object is first accessed. Because System V Interprocess Communication (SVIPC) objects are accessed statelessly, checks are made for every access. For objects with file system names, it is necessary to be able to resolve the name of the actual object. Names are resolved either relatively (to the process' working directory) or absolutely (to the process' root directory). All name resolution begins by searching one of these.

The discretionary access control mechanism allows for effective access control of information resources and provides for separate protection of the confidentiality and integrity of the information. Owner-controlled access control mechanisms are only as effective as users make them. All users must understand how access permissions are granted and denied, and how these are set.

Displaying Access Control Information (aclget Command)

To display the access control information of a file, use the **aclget** command. The information that you view includes attributes, base permissions, and extended permissions.

For example, to display the access control information for the **status** file, type:

```
aclget status
```

Press Enter. The access control information that displays includes a list of attributes, base permissions, and extended permissions. For an example, see "Access Control List Example" on page 125.

See the **aclget** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Setting Access Control Information (aclput Command)

To set the access control information for a file, use the **aclput** command.

> **Note:** The access control list for a file cannot exceed one memory page (approximately 4096 bytes) in size.

For example, to set the access control information for the **status** file with the access control information stored in the `acldefs` file, type:

```
aclput -i acldefs status
```

Press Enter.

For example, to set the access control information for the **status** file with the same information used for the **plans** file, type:

```
aclget plans | aclput status
```

Press Enter.

See the **aclput** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Editing Access Control Information (acledit Command)

To change the access control information of a file, use the **acledit** command. The command displays the current access control information and lets the file owner change it. Before making any changes permanent, the command asks if you want to proceed.

> **Note:** The **EDITOR** environment variable must be specified with a complete path name; otherwise, the **acledit** command will fail.

The access control information that displays includes a list of attributes, base permissions, and extended permissions. For an example, see "Access Control List Example" on page 125.

For example, to edit the access control information of the `plans` file, type:

```
acledit plans
```

Press Enter.

See the **acledit** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Locking Your Terminal (lock or xlock Command)

To lock your terminal, use the **lock** command. The **lock** command requests your password, reads it, and requests the password a second time to verify it. In the interim, the command locks the terminal and does not relinquish it until the password is received the second time. The timeout default value is 15 minutes, but this can be changed with the **-***Number* flag.

Note: If your interface is AIXwindows, use the **xlock** command in the same manner.

For example, to lock your terminal under password control, type:

```
lock
```

Press Enter. You are prompted for the password twice so the system can verify it. If the password is not repeated within 15 minutes, the command times out.

To reserve a terminal under password control with a timeout interval of `10` minutes, type:

```
lock -10
```

Press Enter.

See the **lock** or the **xlock** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Command Summary for File and System Security

acledit	Edits the access control information of a file
aclget	Displays the access control information of a file
aclput	Sets the access control information of a file
chmod	Changes permission modes
chown	Changes the user associated with a file
lock	Reserves a terminal
lsgroup	Displays the attributes of groups
xlock	Locks the local X display until a password is entered

Related Information

"Commands Overview" on page 26

"Processes Overview" on page 35

"File Systems" on page 53

"Directory Overview" on page 56

Chapter 7, "Files" on page 67

Chapter 9, "Backup Files and Storage Media" on page 103

Chapter 11. Customizing the User Environment

The operating system provides various commands and initialization files that enable you to customize the behavior and the appearance of your user environment.

You can also customize some of the default resources of the applications you use on your system. Defaults are initiated by the program at startup. When you change the defaults, you must exit and then restart the program for the new defaults take effect.

For information about customizing the behavior and appearance of the Common Desktop Environment, see the*Common Desktop Environment 1.0: Advanced User's and System Administrator's Guide*.

This chapter discusses the following:

System Startup Files Overview

When you log in, the shell defines your user environment after reading the initialization files that you have set up. The characteristics of your user environment are defined by the values given to your environment variables. You maintain this environment until you log out of the system.

The shell uses two types of profile files when you log in to the operating system. It evaluates the commands contained in the files and then executes the commands to set up your system environment. The files have similar functions except that the **/etc/profile** file controls profile variables for all users on a system whereas the **.profile** file allows you to customize your own environment.

The shell first evaluates the commands contained in the **/etc/profile** file and then runs the commands to set up your system environment in the **/etc/environment** file. After these files are run, the system then checks to see if you have a **.profile** file in your home directory. If the **.profile** file exists, it runs this file. The **.profile** file will specify if there also exists an environment file. If an environment file exists, (usually called **.env**), the system then runs this file and sets up your environment variables.

The **/etc/profile**, **/etc/environment**, and the **.profile** files are run once at login time. The **.env** file, on the other hand, is run every time you open a new shell or a window.

This section discusses the following initialization files:

- "/etc/profile File"
- "/etc/environment File"
- ".profile File" on page 131
- ".env File" on page 131

/etc/profile File

The first file that the operating system uses at login time is the **/etc/profile** file. This file controls systemwide default variables, such as:

- Export variables
- File creation mask (umask)
- Terminal types
- Mail messages to indicate when new mail has arrived

The system administrator configures the **profile** file for all users on the system. Only the system administrator can change this file.

The following example is a typical **/etc/profile** file:

```
#Set file creation mask
unmask 022
#Tell me when new mail arrives
MAIL=/usr/mail/$LOGNAME
#Add my /bin directory to the shell search sequence
PATH=/usr/bin:/usr/sbin:/etc::
#Set terminal type
TERM=lft
#Make some environment variables global
export MAIL PATH TERM
```

See the *AIX 5L Version 5.2 Files Reference* for detailed information about the **/etc/profile** file.

/etc/environment File

The second file that the operating system uses at login time is the **/etc/environment** file. The **/etc/environment** file contains variables specifying the basic environment for all processes. When a new process begins, the **exec** subroutine makes an array of strings available that have the form *Name***=***Value*. This array of strings is called the *environment*. Each name defined by one of the strings is called an *environment variable* or *shell variable*. The **exec** subroutine allows the entire environment to be set at one time.

When you log in, the system sets environment variables from the **environment** file before reading your login profile, named **.profile**. The following variables make up the basic environment:

HOME	The full path name of the user's login or **HOME** directory. The **login** program sets this to the name specified in the **/etc/passwd** file.
LANG	The locale name currently in effect. The **LANG** variable is initially set in the **/etc/profile** file at installation time.
NLSPATH	The full path name for message catalogs.
LOCPATH	The full path name of the location of National Language Support tables.
PATH	The sequence of directories that commands, such as **sh**, **time**, **nice** and **nohup,** search when looking for a command whose path name is incomplete.
TZ	The time zone information. The **TZ** environment variable is initially set by the **/etc/profile** file, the system login profile.

See the *AIX 5L Version 5.2 Files Reference* for detailed information about the **/etc/environment** file.

.profile File

The third file that the operating system uses at login time is the **.profile** file. The **.profile** file is present in your home (**$HOME**) directory and enables you to customize your individual working environment. Because the **.profile** file is hidden, use the **ls -a** command to list it.

After the **login** program adds the **LOGNAME** (login name) and **HOME** (login directory) variables to the environment, the commands in the **$HOME/.profile** file are executed if the file is present. The **.profile** file contains your individual profile that overrides the variables set in the **/etc/profile** file. The **.profile** file is often used to set exported environment variables and terminal modes. You can tailor your environment by modifying the **.profile** file. Use the **.profile** file to control the following defaults:

- Shells to open
- Prompt appearance
- Keyboard sound

The following example is a typical **.profile** file:

```
PATH=/usr/bin:/etc:/home/bin1:/usr/lpp/tps4.0/user::
epath=/home/gsc/e3:
export PATH epath
csh
```

This example has defined two path variables (`PATH` and `epath`), exported them, and opened a C shell (`csh`).

You can also use the **.profile** file (or if it is not present, the **/etc/profile** file) to determine login shell variables. You can also customize other shell environments. For example, use the **.cshrc** file and **.kshrc** file to tailor a C shell and a Korn shell, respectively, when each type of shell is started.

.env File

A fourth file that the operating system uses at login time is the **.env** file, if your **.profile** contains the following line: `export ENV=$HOME/.env`

The **.env** file enables you to customize your individual working environment variables. Because the **.env** file is hidden, use the **ls -a** command to list it. The **.env** file contains the individual user environment variables that override the variables set in the **/etc/environment** file. You can tailor your environment variables as desired by modifying your **.env** file.

The following example is a typical **.env** file:

```
export  myid=`id | sed -n -e 's/).*$//' -e 's/^.*(//p'`
#set prompt: login & system name & path
if [ $myid = root ]
        then    typeset -x PSCH='#:\${PWD}> '
                PS1="#:\${PWD}> "
        else    typeset -x PSCH='>'
                PS1="$LOGNAME@$UNAME:\${PWD}> "
                PS2=">"
                PS3="#?"
fi
export PS1 PS2 PS3
#setup my command aliases
alias   ls="/bin/ls -CF" \
        d="/bin/ls -Fal | pg" \
        rm="/bin/rm -i" \
        up="cd .."
```

Note: When modifying the **.env** file, ensure that newly created environment variables do not conflict with standard variables such as **MAIL**, **PS1**, **PS2**, and **IFS**.

AIXwindows Startup Files Overview

Because different computer systems have different ways of starting the X server and AIXwindows, consult with your system administrator to learn how to get started. Usually, the X server and AIXwindows are started from a shell script that runs automatically when you log in. You might, however, find that you need to start the X server or AIXwindows, or both.

If you log in and find that your display is functioning as a single terminal, with no windows displayed, you can start the X server by typing the following:

```
xinit
```

Press Enter.

If this command does not start the X server, check with your system administrator to ensure that your search path contains the X11 directory containing executable programs. The appropriate path might differ from one system to another.

Note: Before entering this command, make sure that the pointer rests within a window that has a system prompt.

If you log in and find one or more windows without frames, you can start AIXwindows Window Manager by typing the following:

```
mwm &
```

Press Enter.

Because AIXwindows permits customization both by programmers writing AIXwindows applications and by users, you might find that mouse buttons or other functions do not operate as you might expect from reading this documentation. You can reset your AIXwindows environment to the default behavior by pressing and holding the following four keys:

Alt-Ctrl-Shift-!

You can return to the customized behavior by pressing this key sequence again. If your system does not permit this combination of keystrokes, you can also restore default behavior from the default root menu.

.xinitrc File

The **xinit** command uses a customizable shell script file that lists the X client programs to start. The **.xinitrc** file in your home directory controls the windows and applications that start when you start AIXwindows.

The **xinit** command first looks for the **$XINITRC** environment variable to start AIXwindows. If the **$XINITRC** environment variable is not found, it looks for the **$HOME/.xinitrc** shell script. If the **$HOME/.xinitrc** shell script is not found, the **xinit** command starts the **/usr/lib/X11/$LANG/xinitrc** shell script. If **/usr/lib/X11/$LANG/xinitrc** is not found, it looks for the **/usr/lpp/X11/defaults /$LANG/xinitrc** shell script. If that script is not found, it searches for the **/usr/lpp/X11/defaults/xinitrc** shell script.

The **xinitrc** shell script starts commands, such as the **mwm** (AIXwindows Window Manager), **aixterm**, and **xclock** commands.

The **xinit** command performs the following operations:

- Starts an X server on the current display
- Sets up the **$DISPLAY** environment variable
- Runs the **xinitrc** file to start the X client programs

The following example shows the part of the **xinitrc** file you can customize:

```
# This script is invoked by /usr/lpp/X11/bin/xinit

.
.
.
#****************************************************************
#  Start the X clients. Change the following lines to        *
#  whatever command(s) you desire!                          *
#  The default clients are an analog clock (xclock), an lft  *
#  terminal emulator (aixterm), and the Motif Window  Manager  *
#  (mwm).  *
#*****************************************************************
exec mwm
```

.Xdefaults File

If you work in an AIXwindows interface, you can customize this interface with the **.Xdefaults** file. AIXwindows allows you to specify your preferences for visual characteristics, such as colors and fonts.

Many aspects of a windows-based application's appearance and behavior are controlled by sets of variables called *resources*. The visual or behavioral aspect of a resource is determined by its assigned value. There are several different types of values for resources. For example, resources that control color can be assigned predefined values such as *DarkSlateBlue* or *Black*. Resources that specify dimensions are assigned numeric values. Some resources take Boolean values (*True* or *False*).

If you do not have a **.Xdefaults** file in your home directory, you can create one with any text editor. After you have this file in your home directory, you can set resource values in it as you wish. A sample default file called **Xdefaults.tmpl** is in the **/usr/lpp/X11/defaults** directory.

The following example shows part of a typical **.Xdefaults** file:

```
*AutoRaise: on
*DeIconifyWarp: on
*warp:on
*TitleFont:andysans12
*scrollBar: true
*font: Rom10.500
Mwm*menu*foreground: black
Mwm*menu*background: CornflowerBlue
Mwm*menu*RootMenu*foreground: black
Mwm*menu*RootMenu*background: CornflowerBlue
Mwm*icon*foreground: grey25
Mwm*icon*background: LightGray
Mwm*foreground: black
Mwm*background: LightSkyBlue
Mwm*bottomShadowColor: Blue1
Mwm*topShadowColor: CornflowerBlue
Mwm*activeForeground: white
Mwm*activeBackground: Blue1
Mwm*activeBottomShadowColor: black
Mwm*activeTopShadowColor: LightSkyBlue
Mwm*border: black
Mwm*highlight:white

aixterm.foreground: green
aixterm.background: black
aixterm.fullcursor: true
aixterm.ScrollKey: on
aixterm.autoRaise: true
aixterm.autoRaiseDelay: 2
aixterm.boldFont:Rom10.500
aixterm.geometry: 80x25
aixterm.iconFont: Rom8.500
aixterm.iconStartup: false
aixterm.jumpScroll: true
aixterm.reverseWrap: true
aixterm.saveLines: 500
aixterm.scrollInput: true
aixterm.scrollKey: false
aixterm.title: AIX
```

.mwmrc File

Most of the features that you want to customize can be set with resources in your **.Xdefaults** file. However, key bindings, mouse button bindings, and menu definitions for your window manager are specified in the supplementary **.mwmrc** file, which is referenced by resources in the **.Xdefaults** file.

If you do not have a **.mwmrc** file in your home directory, you can copy it as follows:

```
cp /usr/lib/X11/system.mwmrc .mwmrc
```

Because the **.mwmrc** file overrides the systemwide effects of the **system.mwmrc** file, your specifications do not interfere with the specifications of other users.

The following example shows part of a typical **system.mwmrc** file:

```
#  DEFAULT mwm RESOURCE DESCRIPTION FILE (system.mwmrc)
#
# menu pane descriptions
#
# Root Menu Description
Menu RootMenu
{ "Root Menu"        f.title
  no-label           f.separator
  "New Window"       f.exec "aixterm &"
  "Shuffle Up"       f.circle_up
  "Shuffle Down"     f.circle_down
  "Refresh"          f.refresh
  no-label           f.separator
  "Restart"          f.restart
  "Quit"             f.quit_mwm
}
# Default Window Menu Description

Menu DefaultWindowMenu MwmWindowMenu
{ "Restore"   _R   Alt<Key>F5             f.normalize
  "Move"      _M   Alt<Key>F7             f.move
  "Size"      _S   Alt<Key>F8             f.resize
  "Minimize"  _n   Alt<Key>F9             f.minimize
  "Maximize"  _x   Alt<Key>F10            f.maximize
  "Lower"     _L   Alt<Key>F3             f.lower
  no-label                                f.separator
  "Close"     _C   Alt<Key>F4             f.kill
}
# no acclerator window menu
Menu NoAccWindowMenu
{
  "Restore"   _R     f.normalize
  "Move"      _M     f.move
  "Size"      _S     f.resize
  "Minimize"  _n     f.minimize
  "Maximize"  _x     f.maximize
  "Lower"     _L     f.lower
  no-label           f.separator
  "Close"     _C     f.kill
}
Keys DefaultKeyBindings
{
  Shift<Key>Escape             icon|window       f.post_wmenu
  Meta<Key>space               icon|window       f.post_wmenu
  Meta<Key>Tab                 root|icon|window   f.next_key
  Meta Shift<Key>Tab           root|icon|window   f.prev_key
  Meta<Key>Escape              root|icon|window   f.next_key
  Meta Shift<Key>Escape        root|icon|window   f.prev_key
  Meta Ctrl Shift<Key>exclam   root|icon|window   f.set_behavior
}
#
# button binding descriptions
#
Buttons DefaultButtonBindings
{
  <Btn1Down>         frame|icon            f.raise
  <Btn3Down>         frame|icon            f.post_wmenu
  <Btn1Down>         root                  f.menu  RootMenu
  <Btn3Down>         root                  f.menu  RootMenu
  Meta<Btn1Down>     icon|window           f.lower
  Meta<Btn2Down>     window|icon           f.resize
  Meta<Btn3Down>     window                f.move
}
Buttons PointerButtonBindings
{
  <Btn1Down>        frame|icon            f.raise
  <Btn2Down>        frame|icon            f.post_wmenu
  <Btn3Down>        frame|icon            f.lower
  <Btn1Down>        root                  f.menu  RootMenu
  Meta<Btn2Down>    window|icon           f.resize
  Meta<Btn3Down>    window|icon           f.move
}
#
#  END OF mwm RESOURCE DESCRIPTION FILE
#
```

Customization Procedures

This section discusses the following procedures to customize your system environment:

- “Exporting Shell Variables (export Shell Command)”
- “Changing the Display’s Font (chfont Command)” on page 136
- “Changing Control Keys (stty Command)” on page 137
- “Changing Your System Prompt” on page 137

Exporting Shell Variables (export Shell Command)

A *local* shell variable is a variable known only to the shell that created it. If you start a new shell, the old shell’s variables are unknown to it. If you want the new shells that you open to use the variables from an old shell, export the variables to make them *global*.

You can use the **export** command to make local variables global. To make your local shell variables global automatically, export them in your **.profile** file.

Note: Variables can be exported down to child shells, but not exported up to parent shells.

For example, to make the local shell variable PATH global, type:

```
export path
```

Press Enter.

For example, to list all your exported variables, type:

```
export
```

Press Enter.

The system displays information similar to the following:

```
DISPLAY=unix:0
EDITOR=vi
ENV=$HOME/.env
HISTFILE=/u/denise/.history
HISTSIZE=500
HOME=/u/denise
LANG=En_US
LOGNAME=denise
MAIL=/usr/mail/denise
MAILCHECK=0
MAILMSG=**YOU HAVE NEW MAIL.
USE THE mail COMMAND TO SEE YOUR MAILPATH=/usr/mail/denise?denise has mail !!!
MAILRECORD=/u/denise/.Outmail
PATH=/usr/ucb:/usr/lpp/X11/bin:/bin:/usr/bin:/etc:/u/denise:/u/denise/bin:/u/bin1
PWD=/u/denise
SHELL=/bin/ksh
```

Changing the Display's Font (chfont Command)

To change the default font at system startup, use the **chfont** or **smit** command. A *font palette* is a file that the system uses to define and identify the fonts it has available.

Note: To run the **chfont** command, you must have root authority.

chfont Command

For example, to change the active font to the fifth font in the font palette, type:

```
chfont -a5
```

Press Enter. Font ID `5` becomes the primary font.

For example, to change the font to an italic, roman, and bold face of the same size, type:

```
chfont -n /usr/lpp/fonts/It114.snf /usr/lpp/fonts/Bld14.snf  /usr/lpp/fonts/Rom14.snf
```

Press Enter.

See the **chfont** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

smit Command

The **chfont** command can also be run using **smit.**

To select the active font, type:

```
smit chfont
```

Press Enter.

To select the font palette, type:

```
smit chfontpl
```

Press Enter.

Changing Control Keys (stty Command)

To change the keys that your terminal uses for control keys, use the **stty** command. Your changes to control keys last until you log out. To make your changes permanent, place them in your **.profile** file.

For example, to assign Ctrl-Z as the interrupt key, type:

```
stty intr ^Z
```

Be sure to place a space charactor between `intr` and `^Z`. Press Enter.

For example, to reset all control keys to their default values, type:

```
stty sane
```

Press Enter.

For example, to display your current settings, type:

```
stty -a
```

Press Enter.

See the **stty** command in the *AIX 5L Version 5.2 Commands Reference* for the complete syntax.

Changing Your System Prompt

Your shell uses the following prompt variables:

PS1	Prompt used as the normal system prompt
PS2	Prompt used when the shell expects more input
PS3	Prompt used when you have root authority

You can change any of your prompt characters by changing the value of its shell variable. Your prompt changes remain in effect until you log out. To make your changes permanent, place them in your **.env** file.

For example, to display the current value of the PS1 variable, type:

```
echo "prompt is $PS1"
```

Press Enter. The system displays information similar to the following:

```
prompt is $
```

For example, to change your prompt to `Ready>` , type:

```
PS1="Ready> "
```

Press Enter.

For example, to change your continuation prompt to `Enter more->` , type:

```
PS2="Enter more->"
```

Press Enter.

For example, to change your root prompt to `Root->` , type:

```
PS3="Root-> "
```

Press Enter.

Summary for User Environment Customization

System Startup Files

/etc/profile	System file that contains commands that the system executes when you log in
/etc/environment	System file that contains variables specifying the basic environment for all processes
$HOME/.profile	File in your home directory that contains commands that override the system **/etc/profile** when you log in. For more information, see ".profile File" on page 131
$HOME/.env	File in your home directory that overrides the system **/etc/environment** and contains variables specifying the basic environment for all processes. For more information, see ".env File" on page 131

AIXwindows Startup Files

$HOME/.xinitrc	File in your home directory that controls the windows and applications that start up when you start AIXwindows. For more information, see ".xinitrc File" on page 132.
$HOME/.Xdefaults	File in your home directory that controls the visual or behavioral aspect of AIXwindows resources. For more information, see ".Xdefaults File" on page 133.
$HOME/.mwmrc	File in your home directory that defines key bindings, mouse button bindings, and menu definitions for your window manager. For more information, see ".mwmrc File" on page 134.

Customization Procedures

PS1	Normal system prompt
PS2	More input system prompt
PS3	Root system prompt
chfont	Changes the font used by a display at system restart
stty	Sets, resets, and reports workstation operating parameters

Chapter 12. Shells

Your interface to the operating system is called a *shell*. The shell is the outermost layer of the operating system. Shells incorporate a programming language to control processes and files, as well as to start and control other programs. The shell manages the interaction between you and the operating system by prompting you for input, interpreting that input for the operating system, and then handling any resulting output from the operating system.

Shells provide a way for you to communicate with the operating system. This communication is carried out either interactively (input from the keyboard is acted upon immediately) or as a shell script. A *shell script* is a sequence of shell and operating system commands that is stored in a file.

When you log in to the system, the system locates the name of a shell program to execute. After it is executed, the shell displays a command prompt. This prompt is usually a $ (dollar sign). When you type a command at the prompt and press the Enter key, the shell evaluates the command and attempts to carry it out. Depending on your command instructions, the shell writes the command output to the screen or redirects the output. It then returns the command prompt and waits for you to type another command.

A *command line* is the line on which you type. It contains the shell prompt. The basic format for each line is as follows:

```
$ Command Argument(s)
```

The shell considers the first word of a command line (up to the first blank space) as the command, and all subsequent words as arguments.

This chapter discusses the following:

- “Shell Features” on page 140
- “Korn Shell or POSIX Shell Commands” on page 144
- “Quoting in the Korn Shell or POSIX Shell” on page 149
- “Reserved Words in the Korn Shell or POSIX Shell” on page 151
- “Command Aliasing in the Korn Shell or POSIX Shell” on page 151
- “Parameter Substitution in the Korn Shell or POSIX Shell” on page 152
- “Command Substitution in the Korn Shell or POSIX Shell” on page 157
- “Arithmetic Evaluation in the Korn Shell or POSIX Shell” on page 158
- “Field Splitting in the Korn Shell or POSIX Shell” on page 159
- “File-Name Substitution in the Korn Shell or POSIX Shell” on page 159
- “Input and Output Redirection in the Korn Shell or POSIX Shell” on page 161
- “Exit Status in the Korn Shell or POSIX Shell” on page 163
- “Korn Shell or POSIX Shell Commands” on page 144
- “Korn Shell or POSIX Shell Built-In Commands” on page 163
- “Conditional Expressions for the Korn Shell or POSIX Shell” on page 174
- “Job Control in the Korn Shell or POSIX Shell” on page 175
- “Inline Editing in the Korn Shell or POSIX Shell” on page 176
- “List of Korn Shell or POSIX Shell Built-in Commands” on page 173
- “List of Bourne Shell Built-in Commands” on page 199
- “List of C Shell Built-in Commands” on page 219
- “Bourne Shell” on page 221
- “C Shell” on page 221
- “Bourne Shell” on page 184

- “Restricted Shell” on page 185
- “Bourne Shell Commands” on page 186
- “Variable and File-Name Substitution in the Bourne Shell” on page 193
- “Input and Output Redirection in the Bourne Shell” on page 199
- “C Shell” on page 200
- “C Shell Commands” on page 201
- “History Substitution in the C Shell” on page 209
- “Alias Substitution in the C Shell” on page 212
- “Variable and File-Name Substitution in the C Shell” on page 213
- “Environment Variables in the C Shell” on page 216
- “Input and Output Redirection in the C Shell” on page 218
- “Job Control in the C Shell” on page 219

Shell Features

The primary advantages of interfacing to the system through a shell are as follows:

- **Wildcard substitution in file names (pattern-matching)**

  Carries out commands on a group of files by specifying a pattern to match, rather than an actual file name.

  For more information, see the following:
  - “File-Name Substitution in the Korn Shell or POSIX Shell” on page 159
  - “File-Name Substitution in the Bourne Shell” on page 198
  - “File-Name Substitution in the C Shell” on page 214
- **Background processing**

  Sets up lengthy tasks to run in the background, freeing the terminal for concurrent interactive processing.

  For more information, see the **bg** command in the following:
  - “Job Control in the Korn Shell or POSIX Shell” on page 175
  - “C Shell Built-In Commands” on page 202

  Note: The Bourne shell does not support job control.
- **Command aliasing**

  Gives an alias name to a command or phrase. When the shell encounters an alias on the command line or in a shell script, it substitutes the text to which the alias refers.

  For more information, see the following:
  - “Command Aliasing in the Korn Shell or POSIX Shell” on page 151
  - “Alias Substitution in the C Shell” on page 212

  Note: The Bourne shell does not support command aliasing.
- **Command history**

  Records the commands you enter in a history file. You can use this file to easily access, modify, and reissue any listed command.

  For more information, see the **history** command in the following:
  - “Korn Shell or POSIX Shell Command History” on page 149
  - “C Shell Built-In Commands” on page 202
  - “History Substitution in the C Shell” on page 209

Note: The Bourne shell does not support command history.

- **File-name substitution**

  Automatically produces a list of file names on a command line using pattern-matching characters.

  For more information, see the following:
  - "File-Name Substitution in the Korn Shell or POSIX Shell" on page 159
  - "File-Name Substitution in the Bourne Shell" on page 198
  - "File-Name Substitution in the C Shell" on page 214
- **Input and output redirection**

  Redirects input away from the keyboard and redirects output to a file or device other than the terminal. For example, input to a program can be provided from a file and redirected to the printer or to another file.

  For more information, see the following:
  - "Input and Output Redirection in the Korn Shell or POSIX Shell" on page 161
  - "Input and Output Redirection in the Bourne Shell" on page 199
  - "Input and Output Redirection in the C Shell" on page 218
- **Piping**

  Links any number of commands together to form a complex program. The standard output of one program becomes the standard input of the next.

  For more information, see the **pipeline** definition in "Shells Terminology" on page 142.
- **Shell variable substitution**

  Stores data in user-defined variables and predefined shell variables.

  For more information, see the following:
  - "Parameter Substitution in the Korn Shell or POSIX Shell" on page 152
  - "Variable Substitution in the Bourne Shell" on page 193
  - "Variable Substitution in the C Shell" on page 213

Available Shells

The following shells are provided with the operating system:
- Korn shell (started with the **ksh** command)
- Bourne shell (started with the **bsh** command)
- Restricted shell (a limited version of the Bourne shell started with the **Rsh** command)
- POSIX shell (also known as the Korn Shell, and started with the **psh** command)
- Default shell (started with the **sh** command)
- C shell (started with the **csh** command)
- Trusted shell (a limited version of the Korn shell started with the **tsh** command)
- Remote shell (started with the **rsh** command)

The *login shell* refers to the shell that is loaded when you log in to the computer system. Your login shell is set in the **/etc/passwd** file. The Korn shell is the standard operating system login shell and is backwardly compatible with the Bourne Shell (see "Bourne Shell" on page 184).

The *default* or *standard shell* refers to the shell linked to and started with the **/usr/bin/sh** command. The Bourne shell is set up as the default shell and is a subset of the Korn shell.

The **/usr/bin/sh** resides as a copy of the Korn shell, which is **/usr/bin/ksh**. Hence, the Korn shell can be substituted as the default shell. The POSIX shell, which is invoked by the **/usr/bin/psh** command, resides as a link to the**/usr/bin/sh** command.

Shells Terminology

The following definitions are helpful in understanding shells:

blank
A blank is one of the characters in the blank character class defined in the LC_CTYPE category. In the POSIX shell, a blank is either a tab or space.

built-in command
A command that the shell executes without searching for it and creating a separate process.

command
A sequence of characters in the syntax of the shell language. The shell reads each command and carries out the desired action either directly or by invoking separate utilities.

comment
Any word that begins with pound sign (#). The word and all characters that follow it, until the next newline character, are ignored.

identifier
A sequence of letters, digits, or underscores from the portable character set, starting with a letter or underscore. The first character of an identifier must not be a digit. Identifiers are used as names for aliases, functions, and named parameters.

list
A sequence of one or more pipelines separated by one of the following symbols: semicolon (;), ampersand (&), double ampersand (&&), or double bar (||). The list is optionally ended by one of the following symbols: semicolon (;), ampersand (&), or bar ampersand (| &).

; Sequentially processes the preceding pipeline. The shell carries out each command in turn and waits for the most recent command to complete.

& Asynchronously processes the preceding pipeline. The shell carries out each command in turn, processing the pipeline in the background without waiting for it to complete.

|& Asynchronously processes the preceding pipeline and establishes a two-way pipe to the parent shell. The shell carries out each command in turn, processing the pipeline in the background without waiting for it to complete. The parent shell can read from and write to the standard input and output of the spawned command by using the **read -p** and **print -p** commands. Only one such command can be active at any given time.

&& Processes the list that follows this symbol only if the preceding pipeline returns an exit value of zero (0).

|| Processes the list that follows this symbol only if the preceding pipeline returns a nonzero exit value.

The semicolon (;), ampersand (&), and bar ampersand (|&) have a lower priority than the double ampersand (&&) and double bar (||). The ;, &, and |& symbols have equal priority among themselves. The && and || symbols are equal in priority. One or more newline characters can be used instead of a semicolon to delimit two commands in a list.

Note: The |& symbol is valid only in the Korn shell.

metacharacter
Each metacharacter has a special meaning to the shell and causes termination of a word unless it is quoted. Metacharacters are: pipe (|), ampersand (&), semicolon (;), less-than sign (<), greater-than sign (>), left parenthesis ((), right parenthesis ()), dollar sign ($), backquote (`), backslash (\), right quote ('), double quotation marks (″), newline character, space character, and tab character. All characters enclosed between single quotation marks are considered quoted and are interpreted literally by the shell. The special meaning of metacharacters is retained if not quoted. (Metacharacters are also known as *parser metacharacters* in the C shell.)

parameter assignment list
Includes one or more words of the form *Identifier*=*Value* in which spaces surrounding the equal sign (=) must be balanced. That is, leading and trailing blanks, or no blanks, must be used.

Note: In the C shell, the parameter assignment list is of the form **set** *Identifier* = *Value*. The spaces surrounding the equal sign (=) are required.

pipeline	A sequence of one or more commands separated by pipe (|). Each command in the pipeline, except possibly the last command, is run as a separate process. However, the standard output of each command that is connected by a pipe becomes the standard input of the next command in the sequence. If a list is enclosed with parentheses, it is carried out as a simple command that operates in a separate subshell. If the reserved word ! does not precede the pipeline, the exit status will be the exit status of the last command specified in the pipeline. Otherwise, the exit status is the logical NOT of the exit status of the last command. In other words, if the last command returns zero, the exit status will be 1. If the last command returns greater than zero, the exit status will be zero. The format for a pipeline is as follows: `[!] command1 [ | command2 ...]` **Note:** Early versions of the Bourne shell used the caret (^) to indicate a pipe.
shell variable	A name or parameter to which a value is assigned. Assign a variable by typing the variable name, an equal sign (=), and then the value. The variable name can be substituted for the assigned value by preceding the variable name with a dollar sign ($). Variables are particularly useful for creating a short notation for a long path name, such as **$HOME** for the home directory. A predefined variable is one whose value is assigned by the shell. A user-defined variable is one whose value is assigned by a user.
simple command	A sequence of optional parameter assignment lists and redirections, in any sequence. They are optionally followed by commands, words, and redirections. They are terminated by ; , |, &, | |, &&, |&, or a newline character. The command name is passed as parameter 0 (as defined by the **exec** subroutine). The value of a simple command is its exit status of zero if it terminates normally or nonzero if it terminates abnormally. The **sigaction**, **sigvec**, or **signal** Subroutine in the *AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2* includes a list of signal-exit status values.
subshell	A shell that is running as a child of the login shell or the current shell.
wildcard character	Also known as a *pattern-matching character*. The shell associates them with assigned values. The basic wildcards are **?**, *****, [set], and [!set]. Wildcard characters are particularly useful when performing file-name substitution.
word	A sequence of characters that does not contain any blanks. Words are separated by one or more metacharacters.

Creating and Running a Shell Script

Shell scripts provide an easy way to carry out tedious commands, large or complicated sequences of commands, and routine tasks. A shell script is a file that contains one or more commands. When you type the name of a shell script file, the system executes the command sequence contained by the file.

You can create a shell script using a text editor. Your script can contain both operating system commands and shell built-in commands.

The following steps are general guidelines for writing shell scripts:

1. Using a text editor, create and save a file. You can include any combination of shell and operating system commands in the shell script file. By convention, shell scripts that are not set up for use by many users are stored in the **$HOME/bin** directory.

   Note: The operating system does not support the **setuid** or **setgid** subroutines within a shell script.
2. Use the **chmod** command to allow only the owner to run (or execute) the file. For example, if your file is named `script1`, type:
   ```
   chmod u=rwx script1
   ```

Press Enter.

3. Enter the script name on the command line to run the shell script. To run the **script1** shell script, type:

   ```
   script1
   ```

   Press Enter.

   Note: You can run a shell script without making it executable if a shell command (**ksh**, **bsh**, or **csh**) precedes the shell script file name on the command line. For example, to run a nonexecutable file named **script1** under the control of the Korn shell, type:

   ```
   ksh script1
   ```

Specifying a Shell for a Script File

When you run an executable shell script in either the Korn (the POSIX Shell) or Bourne shell, the commands in the script are carried out under the control of the current shell (the shell from which the script is started) unless you specify a different shell. When you run an executable shell script in the C shell, the commands in the script are carried out under the control of the Bourne shell (**/usr/bin/bsh**) unless you specify a different shell.

You can run a shell script in a specific shell by including the shell within the shell script.

To run an executable shell script under a specific shell, type `#!`*`Path`* on the first line of the shell script, and press Enter. The `#!` characters identify the file type. The *Path* variable specifies the path name of the shell from which to run the shell script.

For example, to run the `bsh` script in the Bourne shell, type:

```
#!/usr/bin/bsh
```

Press Enter.

When you precede a shell script file name with a shell command, the shell specified on the command line overrides any shell specified within the script file itself. Therefore, typing `ksh myfile` and pressing Enter runs the file named **myfile** under the control of the Korn shell, even if the first line of **myfile** is `#!/usr/bin/csh`.

Korn Shell or POSIX Shell Commands

The Korn shell is an interactive command interpreter and command programming language. It conforms to the Portable Operating System Interface for Computer Environments (POSIX), an international standard for operating systems. POSIX is not an operating system, but is a *standard* aimed at portability of applications, at the source level, across many systems. POSIX features are built on top of the Korn shell. The Korn shell (also known as the POSIX shell) offers many of the same features as the Bourne and C shells, such as I/O redirection capabilities, variable substitution, and file name substitution. It also includes several additional command and programming language features:

Arithmetic evaluation	The Korn shell, or POSIX shell, can perform integer arithmetic using the built-in **let** command, using any base from 2 to 36. "Arithmetic Evaluation in the Korn Shell or POSIX Shell" on page 158 further describes this feature.
Command history	The Korn shell, or POSIX shell, stores a file that records all of the commands you enter. You can use a text editor to alter a command in this history file and then reissue the command. For more information about the command history feature, see "Korn Shell or POSIX Shell Command History" on page 149.
Coprocess facility	Enables you to run programs in the background and send and receive information to these background processes. For more information, see"Coprocess Facility" on page 162.

Editing	The Korn shell, or POSIX shell, offers inline editing options that enable you to edit the command line. Editors similar to emacs, gmacs, and vi are available. "Inline Editing in the Korn Shell or POSIX Shell" on page 176 further describes this feature.

A Korn shell command is one of the following:
- Simple command
- Pipeline
- List
- Compound command
- Function

When you issue a command in the Korn shell or POSIX shell, the shell evaluates the command and does the following:
- Makes all indicated substitutions.
- Determines whether the command contains a /. If it does, the shell runs the program named by the specified path name.

If the command does not contain a /, the Korn shell or POSIX shell continues with the following actions:
- Determines whether the command is a special built-in command. If it is, the shell runs the command within the current shell process.

  For information about special built-in commands, see ″"Korn Shell or POSIX Shell Built-In Commands" on page 163″.
- Compares the command to user-defined functions. If the command matches a user-defined function, the positional parameters are saved and then reset to the arguments of the **function** call. When the function completes or issues a return, the positional parameter list is restored, and any trap set on **EXIT** within the function is carried out. The value of a function is the value of the last command executed. A function is carried out in the current shell process.
- If the command name matches the name of a regular built-in command, that regular built-in command will be invoked.

  For information about regular built-in commands, see ″"Korn Shell or POSIX Shell Built-In Commands" on page 163″.
- Creates a process and attempts to carry out the command by using the **exec** command (if the command is neither a built-in command nor a user-defined function).

The Korn shell, or POSIX shell, searches each directory in a specified path for an executable file. The **PATH** shell variable defines the search path for the directory containing the command. Alternative directory names are separated with a :. The default path is `/usr/bin:` (specifying the **/usr/bin** directory, and the current directory, in that order). The current directory is specified by two or more adjacent colons, or by a colon at the beginning or end of the path list.

If the file has execute permission but is not a directory or an **a.out** file, the shell assumes that it contains shell commands. The current shell process spawns a subshell to read the file. All nonexported aliases, functions, and named parameters are removed from the file. If the shell command file has read permission, or if the **setuid** or **setgid** bits are set on the file, then the shell runs an agent that sets up the permissions and carries out the shell with the shell command file passed down as an open file. A parenthesized command is run in a subshell without removing nonexported quantities.

This section discusses the following:
- "Korn Shell Compound Commands" on page 146
- "Korn Shell Functions" on page 148
- "Korn Shell or POSIX Shell Built-In Commands" on page 163

- “Conditional Expressions for the Korn Shell or POSIX Shell” on page 174

Korn Shell Compound Commands

A compound command can be a list of simple commands, a pipeline, or it can begin with a reserved word. Most of the time, you will use compound commands such as **if**, **while**, and **for** when you are writing shell scripts.

List of Korn Shell or POSIX Shell Compound Commands

for *Identifier* [**in** *Word* ...] **;do** *List* **;done**	Each time a **for** command is executed, the *Identifier* parameter is set to the next word taken from the **in** *Word* ... list. If the **in** *Word* ... command is omitted, then the **for** command executes the **do** *List* command once for each positional parameter that is set. Execution ends when there are no more words in the list. For more information on positional parameters, refer to ″“Parameter Substitution in the Korn Shell or POSIX Shell” on page 152″.
select *Identifier* [**in** *Word* ...] **;do** *List* **;done**	A **select** command prints on standard error (file descriptor 2) the set of words specified, each preceded by a number. If the **in** *Word* ... command is omitted, then the positional parameters are used instead. The **PS3** prompt is printed and a line is read from the standard input. If this line consists of the number of one of the listed words, then the value of the *Identifier* parameter is set to the word corresponding to this number. If the line read from standard input is empty, the selection list is printed again. Otherwise, the value of the *Identifier* parameter is set to null. The contents of the line read from standard input is saved in the **REPLY** parameter. The *List* parameter is executed for each selection until a break or an end-of-file character is encountered. For more information on positional parameters, refer to ″“Parameter Substitution in the Korn Shell or POSIX Shell” on page 152″.
case *Word* **in** [[**(**] *Pattern* [| *Pattern*] ... **)** *List* **;;**] ... **esac**	A **case** command executes the *List* parameter associated with the first *Pattern* parameter that matches the *Word* parameter. The form of the patterns is the same as that used for file-name substitution.
if *List* **;then** *List* [**elif** *List* **;then** *List*] ... [**;else** *List*] **;fi**	The *List* parameter specifies a list of commands to be run. The shell executes the **if** *List* command first. If a zero exit status is returned, it executes the **then** *List* command. Otherwise, the commands specified by the *List* parameter following the **elif** command are executed. If the value returned by the last command in the **elif** *List* command is zero, the **then** *List* command is executed. If the value returned by the last command in the **then** *List* command is zero, the **else** *List* command is executed. If no commands specified by the *List* parameters are executed for the **else** or **then** command, the **if** command returns a zero exit status.
while *List* **;do** *List* **;done**	
until *List* **;do** *List* **;done**	The *List* parameter specifies a list of commands to be run. The **while** command repeatedly executes the commands specified by the *List* parameter. If the exit status of the last command in the **while** *List* command is zero, the **do** *List* command is executed. If the exit status of the last command in the **while** *List* command is not zero, the loop terminates. If no commands in the **do** *List* command are executed, then the **while** command returns a zero exit status. The **until** command might be used in place of the **while** command to negate the loop termination test.
(*List***)**	The *List* parameter specifies a list of commands to run. The shell executes the *List* parameter in a separate environment. **Note:** If two adjacent open parentheses are needed for nesting, you must insert a space between them in order to differentiate between the command and arithmetic evaluation.

{ *List*;**}**	The *List* parameter specifies a list of commands to run. The *List* parameter is simply executed. **Note:** Unlike the metacharacters ( ), { } denote reserved words (used for special purposes, not as user-declared identifiers). To be recognized, these reserved words must appear at the beginning of a line or after a ;.
[[*Expression*]]	Evaluates the *Expression* parameter. If the expression is true, the command returns a zero exit status.
function *Identifier* **{** *List* **;}** or **function** *Identifier* **()** **{***List* ;**}**	Defines a function that is referred to by the *Identifier* parameter. The body of the function is the specified list of commands enclosed by { }. The () consists of two operators, so mixing blank characters with the *identifier*, (and) is permitted, but is not necessary.
time *Pipeline*	Executes the *Pipeline* parameter. The elapsed time, user time, and system time are printed to standard error.

Shell Startup

You can start the Korn shell with the **ksh** command, **psh** command (POSIX shell), or the **exec** command.

If the shell is started by the **exec** command, and the first character of zero argument (**$0**) is the hyphen (-), then the shell is assumed to be a login shell. The shell first reads commands from the **/etc/profile** file, and then from either the **.profile** file in the current directory or from the **$HOME/.profile** file, if either file exists. Next, the shell reads commands from the file named by performing parameter substitution on the value of the **ENV** environment variable, if the file exists.

If you specify the *File* [*Parameter*] parameter when invoking the Korn shell or POSIX shell, the shell runs the script file identified by the *File* parameter, including any parameters specified. The script file specified must have read permission; any **setuid** and **setgid** settings are ignored. The shell then reads the commands.

Note: Do not specify a script file with the **-c** or **-s** flags when invoking the Korn shell or POSIX shell.

For more information on positional parameters, see “Parameter Substitution in the Korn Shell or POSIX Shell” on page 152.

Korn Shell Environment

All variables (with their associated values) known to a command at the beginning of its execution constitute its *environment*. This environment includes variables that a command inherits from its parent process and variables specified as keyword parameters on the command line that calls the command. The shell interacts with the environment in several ways. When it is started, the shell scans the environment and creates a parameter for each name found, giving the parameter the corresponding value and marking it for export. Executed commands inherit the environment.

If you modify the values of the shell parameters or create new ones using the **export** or **typeset -x** commands, the parameters become part of the environment. The environment seen by any executed command is therefore composed of any name-value pairs originally inherited by the shell, whose values might be modified by the current shell, plus any additions that resulted from using the **export** or **typeset -x** commands. The executed command (subshell) will see any modifications it makes to the environment variables it has inherited, but for its child shells or processes to see the modified values, the subshell must export these variables.

The environment for any simple command or function is changed by prefixing with one or more parameter assignments. A parameter assignment argument is a word of the form *Identifier*=*Value*. Thus, the two following expressions are equivalent (as far as the execution of the command is concerned):

```
TERM=450 Command arguments
(export TERM; TERM=450; Command arguments)
```

Korn Shell Functions

The **function** reserved word defines shell functions. The shell reads and stores functions internally. Alias names are resolved when the function is read. The shell executes functions in the same manner as commands, with the arguments passed as positional parameters. For more information on positional parameters, refer to "Parameter Substitution in the Korn Shell or POSIX Shell" on page 152.

The Korn shell or POSIX shell executes functions in the environment from which functions are invoked. All of the following are shared by the function and the invoking script, and side effects can be produced:

- Variable values and attributes (unless you use **typeset** command within the function to declare a local variable)
- Working directory
- Aliases, function definitions, and attributes
- Special parameter $
- Open files

The following are not shared between the function and the invoking script, and there are no side effects:

- Positional parameters.
- Special parameter #.
- Variables in a variable assignment list when the function is invoked.
- Variables declared using **typeset** command within the function.
- Options.
- Traps. However, signals ignored by the invoking script will also be ignored by the function.

Note: In earlier versions of the Korn shell, traps other than **EXIT** and **ERR** were shared by the function as well as the invoking script.

If trap on **0** or **EXIT** is executed inside the body of a function, the action is executed after the function completes, in the environment that called the function. If the trap is executed outside the body of a function, the action is executed upon exit from the Korn shell. In earlier versions of the Korn shell, no trap on **0** or **EXIT** outside the body of a function was executed upon exit from the function.

When a function is executed, it has the same syntax-error and variable-assignment properties described in ″"Korn Shell or POSIX Shell Built-In Commands" on page 163.

The compound command is executed whenever the function name is specified as the name of a simple command. The operands to the command temporarily will become the positional parameters during the execution of the compound command. The special parameter # will also change to reflect the number of operands. The special parameter 0 will not change.

The **return** special command is used to return from function calls. Errors within functions return control to the caller.

Function identifiers are listed with the **-f** or **+f** option of the **typeset** special command. The **-f** option also lists the text of functions. Functions are undefined with the **-f** option of the **unset** special command.

Ordinarily, functions are unset when the shell executes a shell script. The **-xf** option of the **typeset** special command allows a function to be exported to scripts that are executed without a separate invocation of the shell. Functions that must be defined across separate invocations of the shell should be specified in the **ENV** file with the **-xf** option of the **typeset** special command.

The exit status of a function definition is zero if the function was not successfully declared. Otherwise, it will be greater than zero. The exit status of a function invocation is the exit status of the most recent command executed by the function.

Korn Shell or POSIX Shell Command History

The Korn shell or POSIX shell saves commands entered from your terminal device to a history file. If set, the **HISTFILE** variable value is the name of the history file. If the **HISTFILE** variable is not set or cannot be written, the history file used is **$HOME/.sh_history**. If the history file does not exist and the Korn shell cannot create it, or if it does exist and the Korn shell does not have permission to append to it, then the Korn shell uses a temporary file as the history file. The shell accesses the commands of all interactive shells using the same named history file with appropriate permissions.

By default, the Korn shell or POSIX shell saves the text of the last 128 commands entered from a terminal device. The history file size (specified by the **HISTSIZE** variable) is not limited, although a very large history file can cause the Korn shell to start slowly.

Command History Substitution

Use the **fc** built-in command to list or edit portions of the history file. To select a portion of the file to edit or list, specify the number or the first character or characters of the command. You can specify a single command or range of commands.

If you do not specify an editor program as an argument to the **fc** regular built-in command, the editor specified by the **FCEDIT** variable is used. If the **FCEDIT** variable is not defined, then the **/usr/bin/ed** file is used. The edited command or commands are printed and run when you exit the editor.

The editor name hyphen (-) is used to skip the editing phase and run the command again. In this case, a substitution parameter of the form *Old***=***New* can be used to modify the command before it is run. For example, if `r` is aliased to `fc -e -`, then typing `r bad=good c` runs the most recent command that starts with the letter *c*, and replaces the first occurrence of the `bad` string with the `good` string.

For more information about using the history shell command, see “Listing Previously Entered Commands (history Shell Command)” on page 30 and the **fc** command in the *AIX 5L Version 5.2 Commands Reference*.

Quoting in the Korn Shell or POSIX Shell

When you want the Korn shell or POSIX shell to read a character as a regular character, rather than with any normally associated meaning, you must quote it. To negate the special meaning of a metacharacter, use one of the quoting mechanisms in the following list.

Each metacharacter has a special meaning to the shell and, unless quoted, causes termination of a word. The following characters are considered metacharacters by the Korn shell or POSIX shell and must be quoted if they are to represent themselves:

- pipe (|)
- ampersand (&)
- semicolon (;)
- less-than sign ($lt;) and greater-than sign (>)
- left parenthesis (() and right parenthesis ())
- dollar sign ($)
- backquote (`) and single quotation mark (’)
- backslash (\)
- double-quotation marks (″)

- newline character
- space character
- tab character

The quoting mechanisms are the backslash (\), single quotation mark ('), and double quotation marks (″).

Backslash)
: A backslash (\) that is not quoted preserves the literal value of the following character, with the exception of a newline character. If a new-line character follows the backslash, the shell interprets this as line continuation.

Single Quotation Marks
: Enclosing characters in single quotation marks (' ') preserves the literal value of each character within the single quotation marks. A single quotation mark cannot occur within single quotation marks.

  A backslash cannot be used to escape a single quotation mark in a string that is set in single-quotation marks. An embedded quotation mark can be created by writing, for example: `'a'\''b'`, which yields a'b.

Double Quotation Marks
: Enclosing characters in double quotation marks (″ ″) preserves the literal value of all characters within the double quotation marks, with the exception of the dollar sign, backquote, and backslash characters, as follows:

  $ The dollar sign retains its special meaning introducing parameter expansion, a form of command substitution, and arithmetic expansion.

  The input characters within the quoted string that are also enclosed between $(and the matching) will not be affected by the double quotation marks, but define that command whose output replaces the $(...) when the word is expanded.

  Within the string of characters from an enclosed ${ to the matching }, there must be an even number of unescaped double quotation marks or single quotation marks, if any. A preceding backslash character must be used to escape a literal { or }.

  **`** The backquote retains its special meaning introducing the other form of command substitution. The portion of the quoted string, from the initial backquote and the characters up to the next backquote that is not preceded by a backslash, defines that command whose output replaces ` ... ` when the word is expanded.

  \ The backslash retains its special meaning as an escape character only when followed by one of the following characters: $, `, ″, \, or a newline character.

A double quotation mark must be preceded by a backslash to be included within double quotation marks. When you use double quotation marks, if a backslash is immediately followed by a character that would be interpreted as having a special meaning, the backslash is deleted, and the subsequent character is taken literally. If a backslash does not precede a character that would have a special meaning, it is left in place unchanged, and the character immediately following it is also left unchanged. For example:

```
"\$"    ->      $
"\a"    ->      \a
```

The following conditions apply to metacharacters and quoting characters in the Korn or POSIX shell:

- The meanings of dollar sign, asterisk ($*) and dollar sign, at sign ($@) are identical when not quoted, when used as a parameter assignment value, or when used as a file name.
- When used as a command argument, double quotation marks, dollar sign, asterisk, double quotation marks (″$*″) is equivalent to ″$1*d*$2*d*...″, where *d* is the first character of the IFS parameter.
- Double quotation marks, at sign, asterisk, double quotation marks (″$@″) are equivalent to ″$1″ ″$2″ ....

- Inside backquotes (``), the backslash quotes the characters backslash (\), single quotation mark ('), and dollar sign ($). If the backquotes occur within double quotation marks (" "), the backslash also quotes the double quotation marks character.
- Parameter and command substitution occurs inside double quotation marks (" ").
- The special meaning of reserved words or aliases is removed by quoting any character of the reserved word. You cannot quote function names or built-in command names.

Reserved Words in the Korn Shell or POSIX Shell

The following reserved words have special meaning to the shell:

```
!         case    do
done      elif    else
esac      fi      for
function  if      in
select    then    time
until     while   {
}         [[      ]]
```

The reserved words are recognized only when they appear without quotation marks and when the word is used as the following:

- First word of a command
- First word following one of the reserved words other than **case**, **for**, or **in**
- Third word in a **case** or **for** command (only **in** is valid in this case)

Command Aliasing in the Korn Shell or POSIX Shell

The Korn shell, or POSIX shell, allows you to create aliases to customize commands. The **alias** command defines a word of the form `Name=String` as an alias. When you use an alias as the first word of a command line, the Korn shell checks to see if it is already processing an alias with the same name. If it is, the Korn shell does not replace the alias name. If an alias with the same name is not already being processed, the Korn shell replaces the alias name by the value of the alias.

The first character of an alias name can be any printable character, except the metacharacters. The remaining characters must be the same as for a valid identifier. The replacement string can contain any valid shell text, including the metacharacters.

If the last character of the alias value is a blank, the shell also checks the word following the alias for alias substitution. You can use aliases to redefine special built-in commands, but not to redefine reserved words. Alias definitions are not inherited across invocations of **ksh**. However, if you specify **alias -x**, the alias stays in effect for scripts invoked by name, that do not invoke a separate shell. To export an alias definition and to cause child processes to have access to them, you must specify the **alias -x**, as well as the alias definition in your environment file.

To create, list, and export aliases, use the **alias** command. To remove aliases, use the **unalias** command.

The format for creating an alias is as follows:

```
alias Name=String
```

in which the *Name* parameter specifies the name of the alias and the *String* parameter specifies the value of the alias.

The following exported aliases are predefined by the Korn shell, but can be unset or redefined. It is not recommended that you change them, because this might later confuse anyone who expects the alias to work as predefined by the Korn shell.

```
autoload='typeset -fu'
false='let 0'
functions='typeset -f'
hash='alias -t'
history='fc -l'
integer='typeset -i'
nohup='nohup '
r='fc -e -'
true=':'
type='whence -v'
```

Aliases are not supported on noninteractive invocations of the Korn shell (ksh); for example, in a shell script, or with the **-c** option in **ksh**, as in the following:

```
ksh -c alias
```

For more information about aliasing, see “Creating a Command Alias (alias Shell Command)” on page 33 and the **alias** command in the *AIX 5L Version 5.2 Commands Reference*.

Tracked Aliases

Frequently, aliases are used as shorthand for full path names. One aliasing facility option allows you to automatically set the value of an alias to the full path name of a corresponding command. This special type of alias is a *tracked* alias. Tracked aliases speed execution by eliminating the need for the shell to search the **PATH** variable for a full path name.

The **set -h** command turns on command *tracking* so that each time a command is referenced, the shell defines the value of a tracked alias. This value is undefined each time you reset the **PATH** variable.

These aliases remain tracked so that the next subsequent reference will redefine the value. Several tracked aliases are compiled into the shell.

Tilde Substitution

After the shell performs alias substitution, it checks each word to see if it begins with an unquoted tilde (~). If it does, the shell checks the word, up to the first slash (/), to see if it matches a user name in the **/etc/passwd** file. If the shell finds a match, it replaces the ~ character and the name with the login directory of the matched user. This process is called *tilde substitution*.

The shell does not change the original text if it does not find a match. The Korn shell also makes special replacements if the ~ character is the only character in the word or followed by plus sign (+) or hyphen (-):

~	Replaced by the value of the **HOME** variable.
~+	Replaced by the **$PWD** variable (the full path name of the current directory).
~-	Replaced by the **$OLDPWD** variable (the full path name of the previous directory).

In addition, the shell attempts tilde substitution when the value of a variable assignment parameter begins with a tilde ~ character.

Parameter Substitution in the Korn Shell or POSIX Shell

The Korn Shell, or POSIX shell, enables you to do parameter substitutions.

This section discusses the following:

- “Parameters in the Korn Shell” on page 153
- “Parameter Substitution” on page 153
- “Predefined Special Parameters” on page 154

- "Variables Set by the Korn Shell or POSIX Shell" on page 155
- "Variables Used by the Korn Shell or POSIX Shell" on page 155

Parameters in the Korn Shell

A parameter is defined as the following:

- Identifier of any of the characters asterisk (*), at sign (@), pound sign (#), question mark (?), hyphen (-), dollar sign ($), and exclamation point (!). These are called *special parameters*.
- Argument denoted by a number (*positional parameter*)
- Parameter denoted by an identifier, with a value and zero or more attributes (*named parameter/variables*).

The **typeset** special built-in command assigns values and attributes to named parameters. The attributes supported by the Korn shell are described with the **typeset** special built-in command. Exported parameters pass values and attributes to the environment.

The value of a named parameter is assigned by:

```
Name=Value [ Name=Value ] ...
```

If the **-i** integer attribute is set for the *Name* parameter, the *Value* parameter is subject to arithmetic evaluation. Refer to "Arithmetic Evaluation in the Korn Shell or POSIX Shell" on page 158 for more information about arithmetic expression evaluation.

The shell supports a one-dimensional array facility. An element of an array parameter is referenced by a subscript. A subscript is denoted by an arithmetic expression enclosed by brackets ([ ]). To assign values to an array, use `set -A Name Value` ... . The value of all subscripts must be in the range of 0 through 511. Arrays need not be declared. Any reference to a named parameter with a valid subscript is legal and an array will be created, if necessary. Referencing an array without a subscript is equivalent to referencing the element 0.

Positional parameters are assigned values with the **set** special command. The **$0** parameter is set from argument 0 when the shell is invoked. The $ character is used to introduce parameters that can be substituted.

Parameter Substitution

The following are substitutable parameters:

${*Parameter***}**	The shell reads all the characters from the **${** to the matching **}** as part of the same word, even if that word contains braces or metacharacters. The value, if any, of the specified parameter is substituted. The braces are required when the *Parameter* parameter is followed by a letter, digit, or underscore that is not to be interpreted as part of its name, or when a named parameter is subscripted. If the specified parameter contains one or more digits, it is a *positional parameter*. A positional parameter of more than one digit must be enclosed in braces. If the value of the variable is an ***** or an **@**), each positional parameter, starting with **$1**, is substituted (separated by a field separator character). If an array identifier with a subscript ***** or an **@** is used, then the value for each of the elements (separated by a field separator character) is substituted.
${#*Parameter***}**	If the value of the *Parameter* parameter is an * or an @, the number of positional parameters is substituted. Otherwise, the length specified by the *Parameter* parameter is substituted.
${#*Identifier***[*]}**	The number of elements in the array specified by the *Identifier* parameter is substituted.

${*Parameter***:-***Word***}**	If the *Parameter* parameter is set and is not null, then its value is substituted; otherwise, the value of the *Word* parameter is substituted.
${*Parameter***:=***Word***}**	If the *Parameter* parameter is not set or is null, then it is set to the value of the *Word* parameter. Positional parameters cannot be assigned in this way.
${*Parameter***:?***Word***}**	If the *Parameter* parameter is set and is not null, then substitute its value. Otherwise, print the value of the *Word* variable and exit from the shell. If the *Word* variable is omitted, then a standard message is printed.
${*Parameter***:+***Word***}**	If the *Parameter* parameter is set and is not null, then substitute the value of the *Word* variable.
${***Parameter***#***Pattern***}** \| **${*Parameter***##***Pattern***}**	If the specified shell *Pattern* parameter matches the beginning of the value of the *Parameter* parameter, then the value of this substitution is the value of the *Parameter* parameter with the matched portion deleted. Otherwise, the value of the *Parameter* parameter is substituted. In the first form, the smallest matching pattern is deleted. In the second form, the largest matching pattern is deleted.
${***Parameter***%***Pattern***}** \| **${*Parameter***%%***Pattern***}**	If the specified shell *Pattern* matches the end of the value of the *Parameter* variable, then the value of this substitution is the value of the *Parameter* variable with the matched part deleted. Otherwise, substitute the value of the *Parameter* variable. In the first form, the smallest matching pattern is deleted; in the second form, the largest matching pattern is deleted.

In the previous expressions, the *Word* variable is not evaluated unless it is to be used as the substituted string. Thus, in the following example, the **pwd** command is executed only if the **-d** flag is not set or is null:

```
echo ${d:-$(pwd)}
```

Note: If the **:** is omitted from the previous expressions, the shell checks only whether the *Parameter* parameter is set.

Predefined Special Parameters

The following parameters are automatically set by the shell:

@	Expands the positional parameters, beginning with **$1**. Each parameter is separated by a space. If you place ″ around **$@**, the shell considers each positional parameter a separate string. If no positional parameters exist, the shell expands the statement to an unquoted null string.
*****	Expands the positional parameters, beginning with **$1**. The shell separates each parameter with the first character of the **IFS** parameter value. If you place ″ around **$***, the shell includes the positional parameter values in double quotation marks. Each value is separated by the first character of the **IFS** parameter.
#	Specifies the number (in decimals) of positional parameters passed to the shell, not counting the name of the shell procedure itself. The **$#** parameter thus yields the number of the highest-numbered positional parameter that is set. One of the primary uses of this parameter is to check for the presence of the required number of arguments.
-	Supplies flags to the shell on invocation or with the **set** command.
?	Specifies the exit value of the last command executed. Its value is a decimal string. Most commands return 0 to indicate successful completion. The shell itself returns the current value of the **$?** parameter as its exit value.

$	Identifies the process number of this shell. Because process numbers are unique among all existing processes, this string of up to 5 digits is often used to generate unique names for temporary files.
	
	The following example illustrates the recommended practice of creating temporary files in a directory used only for that purpose:
	`temp=$HOME/temp/$$`
	`ls >$temp`
	`.`
	`.`
	`.`
	`rm $temp`

!	Specifies the process number of the most recent background command invoked.
zero (0)	Expands to the name of the shell or shell script.

Variables Set by the Korn Shell or POSIX Shell

The following variables are set by the shell:

underscore (_)	Indicates initially the absolute path name of the shell or script being executed as passed in the environment. Subsequently, it is assigned the last argument of the previous command. This parameter is not set for commands that are asynchronous. This parameter is also used to hold the name of the matching **MAIL** file when checking for mail.
ERRNO	Specifies a value that is set by the most recently failed subroutine. This value is system-dependent and is intended for debugging purposes.
LINENO	Specifies the line number of the current line within the script or function being executed.
OLDPWD	Indicates the previous working directory set by the **cd** command.
OPTARG	Specifies the value of the last option argument processed by the **getopts** regular built-in command.
OPTIND	Specifies index of the last option argument processed by the **getopts** regular built-in command.
PPID	Identifies the process number of the parent of the shell.
PWD	Indicates the present working directory set by the **cd** command.
RANDOM	Generates a random integer, uniformly distributed between 0 and 32767. The sequence of random numbers can be initialized by assigning a numeric value to the **RANDOM** variable.
REPLY	Set by the **select** statement and by the **read** regular built-in command when no arguments are supplied.
SECONDS	Specifies the number of seconds since shell invocation is returned. If this variable is assigned a value, then the value returned upon reference will be the value that was assigned plus the number of seconds since the assignment.

Variables Used by the Korn Shell or POSIX Shell

The following variables are used by the shell:

CDPATH	Indicates the search path for the **cd** (change directory) command.
COLUMNS	Defines the width of the edit window for the shell edit modes and for printing **select** lists.
EDITOR	If the value of this parameter ends in emacs, gmacs, or vi, and the **VISUAL** variable is not set with the **set** special built-in command, then the corresponding option is turned on.
ENV	If this variable is set, then parameter substitution is performed on the value to generate the path name of the script that will be executed when the shell is invoked. This file is typically used for alias and function definitions.
FCEDIT	Specifies the default editor name for the **fc** regular built-in command.

FPATH	Specifies the search path for function definitions. This path is searched when a function with the **-u** flag is referenced and when a command is not found. If an executable file is found, then it is read and executed in the current environment.
HISTFILE	If this variable is set when the shell is invoked, then the value is the path name of the file that will be used to store the command history.
HISTSIZE	If this variable is set when the shell is invoked, then the number of previously entered commands that are accessible by this shell will be greater than or equal to this number. The default is 128.
HOME	Indicates the name of your login directory, which becomes the current directory upon completion of a login. The **login** program initializes this variable. The **cd** command uses the value of the **$HOME** parameter as its default value. Using this variable rather than an explicit path name in a shell procedure allows the procedure to be run from a different directory without alterations.
IFS	Specifies IFS (internal field separators), normally space, tab, and newline, used to separate command words that result from command or parameter substitution and for separating words with the regular built-in command **read**. The first character of the **IFS** parameter is used to separate arguments for the **$*** substitution.
LANG	Provides a default value for the **LC_*** variables.
LC_ALL	Overrides the value of the **LANG** and **LC_*** variables.
LC_COLLATE	Determines the behavior of range expression within pattern matching.
LC_CTYPE	Defines character classification, case conversion, and other character attributes.
LC_MESSAGES	Determines the language in which messages are written.
LINES	Determines the column length for printing select lists. Select lists print vertically until about two-thirds of lines specified by the **LINES** variable are filled.
MAIL	Specifies the file path name used by the mail system to detect the arrival of new mail. If this variable is set to the name of a mail file and the **MAILPATH** variable is not set, then the shell informs the user of new mail in the specified file.
MAILCHECK	Specifies how often (in seconds) the shell checks for changes in the modification time of any of the files specified by the **MAILPATH** or **MAIL** variables. The default value is 600 seconds. When the time has elapsed, the shell checks before issuing the next prompt.
MAILPATH	Specifies a list of file names separated by colons. If this variable is set, then the shell informs the user of any modifications to the specified files that have occurred during the period, in seconds, specified by the **MAILCHECK** variable. Each file name can be followed by a **?** and a message that will be printed. The message will undergo variable substitution with the **$_** variable defined as the name of the file that has changed. The default message is `you have mail in $_`.
NLSPATH	Determines the location of message catalogs for the processing of **LC_MESSAGES**.
PATH	Indicates the search path for commands, which is an ordered list of directory path names separated by colons. The shell searches these directories in the specified order when it looks for commands. A null string anywhere in the list represents the current directory.
PS1	Specifies the string to be used as the primary system prompt. The value of this parameter is expanded for parameter substitution to define the primary prompt string, which is a **$** by default. The **!** character in the primary prompt string is replaced by the command number.
PS2	Specifies the value of the secondary prompt string, which is a **>** by default.
PS3	Specifies the value of the selection prompt string used within a **select** loop, which is **#?** by default.
PS4	The value of this variable is expanded for parameter substitution and precedes each line of an execution trace. If omitted, the execution trace prompt is a **+**.
SHELL	Specifies the path name of the shell, which is kept in the environment.
SHELL PROMPT	When used interactively, the shell prompts with the value of the **PS1** parameter before reading a command. If at any time a new line is entered and the shell requires further input to complete a command, the shell issues the secondary prompt (the value of the **PS2** parameter).

TMOUT	Specifies the number of seconds a shell waits inactive before exiting. If the **TMOUT** variable is set to a value greater than zero (0), the shell exits if a command is not entered within the prescribed number of seconds after issuing the **PS1** prompt. (Note that the shell can be compiled with a maximum boundary that cannot be exceeded for this value.) **Note:** After the timeout period has expired, there is a 60-second pause before the shell exits.
VISUAL	If the value of this variable ends in emacs, gmacs, or vi, then the corresponding option is turned on.

The shell gives default values to the **PATH**, **PS1**, **PS2**, **MAILCHECK**, **TMOUT**, and **IFS** parameters, but the **HOME**, **SHELL**, **ENV**, and **MAIL** parameters are *not* set by the shell (although the **HOME** parameter is set by the **login** command).

Command Substitution in the Korn Shell or POSIX Shell

The Korn Shell, or POSIX Shell, enables you to do command substitution.

In command substitution, the shell executes a specified command in a subshell environment and replaces that command with its output. To execute command substitution in the Korn shell or POSIX shell, perform the following:

```
$(command)
```

or, for the backquoted version, use:

```
`command`
```

Note: Although the backquote syntax is accepted by **ksh**, it is considered obsolete by the X/Open Portability Guide Issue 4 and POSIX standards. These standards recommend that portable applications use the `$(command)` syntax.

The shell expands the command substitution by executing *command* in a subshell environment and replacing the command substitution (the text of *command* plus the enclosing $( ) or backquotes) with the standard output of the command, removing sequences of one or more newline characters at the end of the substitution.

In the following example, the $( ) surrounding the command indicates that the output of the **whoami** command is substituted:

```
echo My name is: $(whoami)
```

You can perform the same command substitution with:

```
echo My name is: `whoami`
```

The output from both examples for user `dee` is:

```
My name is: dee
```

You can also substitute arithmetic expressions by enclosing them in ( ). For example, the command:

```
echo Each hour contains $((60 * 60)) seconds
```

produces the following result:

```
Each hour contains 3600 seconds
```

The Korn shell or POSIX shell removes all trailing newline characters when performing command substitution. For example, if your current directory contains the `file1`, `file2`, and `file3` files, the command:

```
echo $(ls)
```

removes the newline characters and produces the following output:

```
file1 file2 file3
```

To preserve newline characters, insert the substituted command in ″ ″:

```
echo "$(ls)"
```

Arithmetic Evaluation in the Korn Shell or POSIX Shell

The Korn shell or POSIX shell regular built-in **let** command enables you to perform integer arithmetic. Constants are of the form [*Base*]*Number*. The *Base* parameter is a decimal number between 2 and 36 inclusive, representing the arithmetic base. The *Number* parameter is a number in that base. If you omit the *Base* parameter, the shell uses a base of 10.

Arithmetic expressions use the same syntax, precedence, and associativity of expression as the C language. All of the integral operators, other than double plus (**++**), double hyphen (**—**), question mark, colon (**?:**), and comma (**,**), are supported. The following table lists valid Korn shell or POSIX shell operators in decreasing order of precedence:

Operator	Definition
-	Unary minus
!	Logical negation
~	Bitwise negation
*	Multiplication
/	Division
%	Remainder
+	Addition
-	Subtraction
<<, >>	Left shift, right shift
<=,>=, <>, ==, !=	Comparison
&	Bitwise AND
^	Bitwise exclusive OR
|	Bitwise OR
&&	Logical AND
||	Logical OR
= *=, /=, &= +=, -=, <<=, > >=, &=, ^=, |=	Assignment

Many arithmetic operators, such as *, &, <, and >, have special meaning to the Korn shell or POSIX shell. These characters must be quoted. For example, to multiply the current value of `y` by `5` and reassign the new value to `y`, use the expression:

```
let "y = y * 5"
```

Enclosing the expression in quotation marks removes the special meaning of the ***** character.

You can group operations inside **let** command expressions to force grouping. For example, in the expression:

```
let "z = q * (z - 10)"
```

the command multiplies `q` by the reduced value of `z`.

The Korn shell or POSIX shell includes an alternative form of the **let** command if only a single expression is to be evaluated. The shell treats commands enclosed in **(( ))** as quoted expressions. Therefore, the expression:

```
((x = x / 3))
```

is equivalent to:

```
let "x = x / 3"
```

Named parameters are referenced by name within an arithmetic expression without using the parameter substitution syntax. When a named parameter is referenced, its value is evaluated as an arithmetic expression.

Specify an internal integer representation of a named parameter with the **-i** flag of the **typeset** special built-in command. Using the **-i** flag, arithmetic evaluation is performed on the value of each assignment to a named parameter. If you do not specify an arithmetic base, the first assignment to the parameter determines the arithmetic base. This base is used when parameter substitution occurs.

Field Splitting in the Korn Shell or POSIX Shell

After performing command substitution, the Korn shell scans the results of substitutions for those field separator characters found in the **IFS** (Internal Field Separator) variable. Where such characters are found, the shell splits the substitutions into distinct arguments. The shell retains explicit null arguments ("" or '') and removes implicit null arguments (those resulting from parameters that have no values).

- If the value of **IFS** is a space, tab and newline character, or if it is not set, any sequence of space, tab and newline characters at the beginning or end of the input will be ignored and any sequence of those characters within the input will delimit a field. For example, the following input yields two fields, **school** and **days**:

  ```
  <newline><space><tab>school<tab><tab>days<space>
  ```
- Otherwise, and if the value of **IFS** is not null, the following rules apply in sequence. **IFS** *white space* is used to mean any sequence (zero or more instances) of white-space characters that are in the **IFS** value (for example, if **IFS** contains space/comma/tab, any sequence of space and tab characters is considered **IFS** white space).
  1. **IFS** white space is ignored at the beginning and end of the input.
  2. Each occurrence in the input of an **IFS** character that is not **IFS** white space, along with any adjacent **IFS** white space, delimits a field.
  3. Non-zero length **IFS** white space delimits a field.

File-Name Substitution in the Korn Shell or POSIX Shell

The Korn shell, or POSIX shell, performs file-name substitution by scanning each command word specified by the *Word* variable for certain characters. If a command word includes the *****), **?** or **[** characters, and the **-f** flag has not been set, the shell regards the word as a pattern. The shell replaces the word with file names, sorted according to the collating sequence in effect in the current locale, that match that pattern. If the shell does not find a file name to match the pattern, it does not change the word.

When the shell uses a pattern for file-name substitution, the **.** and **/** characters must be matched explicitly.

> **Note:** The Korn shell does not treat these characters specially in other instances of pattern matching.

These pattern-matching characters indicate the following substitutions:

*****	Matches any string, including the null string.
?	Matches any single character.
[...]	Matches any one of the enclosed characters. A pair of characters separated by a **-** matches any character lexically within the inclusive range of that pair, according to the collating sequence in effect in the current locale. If the first character following the opening **[** is an **!**, then any character not enclosed is matched. A **-** can be included in the character set by putting it as the first or last character.

You can also use the `[:charclass:]` notation to match file names within a range indication. This format instructs the system to match any single character belonging to `class`. The definition of which characters constitute a specific character class is present through the **LC_CTYPE** category of the **setlocale** subroutine. All character classes specified in the current locale are recognized.

The names of some of the character classes are as follows:
- **alnum**
- **alpha**
- **cntrl**
- **digit**
- **graph**
- **lower**
- **print**
- **punct**
- **space**
- **upper**
- **xdigit**

For example, `[[:upper:]]` matches any uppercase letter.

The Korn shell supports file-name expansion based on collating elements, symbols, or equivalence classes.

A *PatternList* is a list of one or more patterns separated from each other with a **|**. Composite patterns are formed with one or more of the following:

?(*PatternList***)**	Optionally matches any one of the given patterns
(PatternList***)**	Matches zero or more occurrences of the given patterns
+(*PatternList***)**	Matches one or more occurrences of the given patterns
@(*PatternList***)**	Matches exactly one of the given patterns
!(*PatternList***)**	Matches anything, except one of the given patterns

Pattern matching has some restrictions. If the first character of a file name is a dot (.), it can be matched only by a pattern that also begins with a dot. For example, ***** matches the file names `myfile` and `yourfile` but not the file names `.myfile` and `.yourfile`. To match these files, use a pattern such as the following:

```
.*file
```

If a pattern does not match any file names, then the pattern itself is returned as the result of the attempted match.

File and directory names should not contain the characters *, ? , [, or] because they can cause infinite recursion (that is, infinite loops) during pattern-matching attempts.

Quote Removal

The quote characters, backslash (\), single quote (’), and double quote (″) that were present in the original word will be removed unless they have themselves been quoted.

Input and Output Redirection in the Korn Shell or POSIX Shell

Before the Korn shell executes a command, it scans the command line for redirection characters. These special notations direct the shell to redirect input and output. Redirection characters can appear anywhere in a simple command or can precede or follow a command. They are not passed on to the invoked command.

The shell performs command and parameter substitution before using the *Word* or *Digit* parameter except as noted. File-name substitution occurs only if the pattern matches a single file and blank interpretation is not performed.

<*Word*	Uses the file specified by the *Word* parameter as standard input (file descriptor 0).
>*Word*	Uses the file specified by the *Word* parameter as standard output (file descriptor 1). If the file does not exist, the shell creates it. If the file exists and the **noclobber** option is on, an error results; otherwise, the file is truncated to zero length.
>|*Word*	Same as the **>***Word* command, except that this redirection statement overrides the **noclobber** option.
> >*Word*	Uses the file specified by the *Word* parameter as standard output. If the file currently exists, the shell appends the output to it (by first seeking the end-of-file character). If the file does not exist, the shell creates it.
<>*Word*	Opens the file specified by the *Word* parameter for reading and writing as standard input.
<<[**-**]*Word*	Reads each line of shell input until it locates a line containing only the value of the *Word* parameter or an end-of-file character. The shell does not perform parameter substitution, command substitution, or file name substitution on the file specified. The resulting document, called a *here document*, becomes the standard input. For more information on here documents, see ″“Using Inline Input (Here) Documents” on page 48″. If any character of the *Word* parameter is quoted, no interpretation is placed upon the characters of the document.

The here document is treated as a single word that begins after the next newline character and continues until there is a line containing only the delimiter, with no trailing blank characters. Then the next here document, if any, starts. The format is as follows:

```
[n]<<word
   here document
delimiter
```

If any character in *word* is quoted, the delimiter is formed by removing the quote on *word*. The here document lines will not be expanded. Otherwise, the delimiter is the *word* itself. If no characters in *word* are quoted, all lines of the here document will be expanded for parameter expansion, command substitution, and arithmetic expansion.

The shell performs parameter substitution for the redirected data. To prevent the shell from interpreting the \, $, and single quotation mark (’) characters and the first character of the *Word* parameter, precede the characters with a \ character.

If a - is appended to <<, the shell strips all leading tabs from the *Word* parameter and the document.

<&*Digit*	Duplicates standard input from the file descriptor specified by the *Digit* parameter
>& *Digit*	Duplicates standard output in the file descriptor specified by the *Digit* parameter
<&-	Closes standard input
>&-	Closes standard output
<&p	Moves input from the coprocess to standard input

>&p	Moves output to the coprocess to standard output

If one of these redirection options is preceded by a digit, then the file descriptor number referred to is specified by the digit (instead of the default 0 or 1). In the following example, the shell opens file descriptor 2 for writing as a duplicate of file descriptor 1:

```
... 2>&1
```

The order in which redirections are specified is significant. The shell evaluates each redirection in terms of the (*FileDescriptor*, *File*) association at the time of evaluation. For example, in the statement:

```
... 1>File 2>&1
```

the file descriptor 1 is associated with the file specified by the *File* parameter. The shell associates file descriptor 2 with the file associated with file descriptor 1 (*File*). If the order of redirections were reversed, file descriptor 2 would be associated with the terminal (assuming file descriptor 1 had previously been) and file descriptor 1 would be associated with the file specified by the *File* parameter.

If a command is followed by an ampersand (&) and job control is not active, the default standard input for the command is the empty file, **/dev/null**. Otherwise, the environment for the execution of a command contains the file descriptors of the invoking shell as modified by input and output specifications.

For more information about redirection, see Chapter 5, “Input and Output Redirection” on page 45.

Coprocess Facility

The Korn shell, or POSIX shell, allows you to run one or more commands as background processes. These commands, run from within a shell script, are called *coprocesses*.

Designate a coprocess by placing the **|&** operator after a command. Both standard input and output of the command are piped to your script.

A coprocess must meet the following restrictions:
- Include a newline character at the end of each message
- Send each output message to standard output
- Clear its standard output after each message

The following example demonstrates how input is passed to and returned from a coprocess:

```
echo "Initial process"
./FileB.sh |&
read -p a b c d
echo "Read from coprocess: $a $b $c $d"
print -p "Passed to the coprocess"
read -p a b c d
echo "Passed back from coprocess: $a $b $c $d"
```

```
FileB.sh
   echo "The coprocess is running"
   read a b c d
   echo $a $b $c $d
```

The resulting standard output is as follows:

```
Initial process
Read from coprocess: The coprocess is running
Passed back from coprocess: Passed to the coprocess
```

To write to the coprocess, use the **print -p** command. To read from the coprocess, use the **read -p** command.

Redirecting Coprocess Input and Output

The standard input and output of a coprocess is reassigned to a numbered file descriptor by using I/O redirection. For example, the command:

```
exec 5>&p
```

moves the input of the coprocess to file descriptor 5.

After this has completed, you can use standard redirection syntax to redirect command output to the coprocess. You can also start another coprocess. Output from both coprocesses is connected to the same pipe and is read with the **read -p** command. To stop the coprocess, type:

```
read -u5
```

Exit Status in the Korn Shell or POSIX Shell

Errors detected by the shell, such as syntax errors, cause the shell to return a nonzero exit status. Otherwise, the shell returns the exit status of the last command carried out. The shell reports detected run-time errors by printing the command or function name and the error condition. If the number of the line on which an error occurred is greater than 1, then the line number is also printed in [ ] (brackets) after the command or function name.

For a noninteractive shell, an error encountered by a special built-in or other type of command will cause the shell to write a diagnostic message as shown in the following table:

Error	Special Built-In	Other Utilities
Shell language syntax error	will exit	will exit
Utility syntax error (option or operand error)	will exit	will not exit
Redirection error	will exit	will not exit
Variable assignment error	will exit	will not exit
Expansion error	will exit	will exit
Command not found	not applicable	may exit
Dot script not found	will exit	not applicable

If any of the errors shown as ″will (may) exit″ occur in a subshell, the subshell will (may) exit with a nonzero status, but the script containing the subshell will not exit because of the error.

In all cases shown in the table, an interactive shell will write a diagnostic message to standard error, without exiting.

Korn Shell or POSIX Shell Built-In Commands

Special commands are built in to the Korn shell and POSIX shell and executed in the shell process. Unless otherwise indicated, the output is written to file descriptor 1 and the exit status is zero (0) if the command does not contain any syntax errors. Input and output redirection is permitted. There are two types of built-in commands, *special built-in commands* and *regular built-in commands*.

Special built-in commands differ from regular built-in commands in the following ways:

- A syntax error in a special built-in command might cause the shell executing the command to end. This does not happen if you have a syntax error in a regular built-in command. If a syntax error in a special built-in command does not end the shell program, the exit value is nonzero.

- Variable assignments specified with special built-in commands remain in effect after the command completes.
- I/O redirections are processed after parameter assignments.

In addition, words that are in the form of a parameter assignment following the **export**, **readonly**, and **typeset** special commands are expanded with the same rules as a parameter assignment. Tilde substitution is performed after the =, and word-splitting and file-name substitution are not performed.

For an alphabetical listing of these commands, refer to the "List of Korn Shell or POSIX Shell Built-in Commands" on page 173

Special Built-in Command Descriptions

The Korn Shell provides the following special built-in commands:

```
:               eval            newgrp          shift
.               exec            readonly        times
break           exit            return          trap
continue        export          set             typeset
                                                unset
```

: [*Argument* ...]	Expands only arguments. It is used when a command is necessary, as in the *then* condition of an **if** command, but nothing is to be done by the command.
. *File* [*Argument* ...]	Reads the complete specified file and then executes the commands. The commands are executed in the current shell environment. The search path specified by the **PATH** variable is used to find the directory containing the specified file. If any arguments are specified, they become the positional parameters. Otherwise, the positional parameters are unchanged. The exit status is the exit status of the most recent command executed. Refer to "Parameter Substitution in the Korn Shell or POSIX Shell" on page 152 for more information on positional parameters. **Note:** The *.File* [*Argument* ...] command reads the entire file before any commands are carried out. Therefore, the **alias** and **unalias** commands in the file do not apply to any functions defined in the file.
break [*n*]	Exits from the enclosing **for**, **while**, **until**, or **select** loop, if one exists. If you specify the *n* parameter, the command breaks the number of levels specified by the *n* parameter. The value of *n* is any integer equal to or greater than 1.
continue [*n*]	Resumes the next iteration of the enclosing **for**, **while**, **until**, or **select** loop. If you specify the *n* variable, the command resumes at the *n*th enclosing loop. The value of *n* is any integer equal to or greater than 1.
eval [*Argument* ...]	Reads the specified arguments as input to the shell and executes the resulting command or commands.
exec [*Argument* ...]	Executes the command specified by the argument in place of this shell (without creating a new process). Input and output arguments can appear and affect the current process. If you do not specify an argument, the **exec** command modifies file descriptors as prescribed by the input and output redirection list. In this case, any file descriptor numbers greater than 2 that are opened with this mechanism are closed when invoking another program.
exit [*n*]	Exits the shell with the exit status specified by the *n* parameter. The *n* parameter must be an unsigned decimal integer with range 0-255. If you omit the *n* parameter, the exit status is that of the most recent command executed. An end-of-file character also exits the shell, unless the **ignoreeof** option of the **set** special command is turned on.
export -p [*Name*[**=** *Value*]] ...	Marks the specified names for automatic export to the environment of subsequently executed commands. **-p** writes to standard output the names and values of all exported variables, in the following format: `"export %s= %s\n", <name> <value>`

newgrp [*Group*]	Equivalent to the **exec/usr/bin/newgrp** [*Group*] command. **Note:** This command does not return.
readonly -p [*Name*[= *Value*]] ...	Marks the names specified by the *Name* parameter as read-only. These names cannot be changed by subsequent assignment. **-p** writes to standard output the names and values of all exported variables, in the following format: `"export %s= %s\n", <name> <value>`
return [*n*]	Causes a shell function to return to the invoking script. The return status is specified by the *n* variable. If you omit the *n* variable, the return status is that of the most recent command executed. If you invoke the **return** command outside of a function or a script, then it is the same as an **exit** command.
set [+ |**-abCefhkmnostuvx**] [+ |**-o** *Option*]... [+ |**-A** *Name*] [*Argument* ...]	If no options or arguments are specified, the **set** command writes the names and values of all shell variables in the collation sequence of the current locale. When options are specified, they will set or unset attributes of the shell, described as follows:

-A	Array assignment. Unsets the *Name* parameter and assigns values sequentially from the specified *Argument* parameter list. If the **+A** flag is used, the *Name* parameter is not unset first.
-a	Exports automatically all subsequent parameters that are defined.
-b	Notifies the user asynchronously of background job completions.
-C	Equivalent to `set -o noclobber`.
-e	Executes the **ERR** trap, if set, and exits if a command has a nonzero exit status. This mode is disabled while reading profiles.
-f	Disables file name substitution.
-h	Designates each command as a tracked alias when first encountered.
-k	Places all parameter-assignment arguments in the environment for a command, not only those arguments that precede the command name.
-m	Runs background jobs in a separate process and prints a line upon completion. The exit status of background jobs is reported in a completion message. On systems with job control, this flag is turned on automatically for interactive shells. For more information, see "Job Control in the Korn Shell or POSIX Shell" on page 175.
-n	Reads commands and checks them for syntax errors, but does not execute them. This flag is ignored for interactive shells.

-o *Option*
Prints current option settings and an error message if you do not specify an argument. You can set more than one option on a single **ksh** command line. If the **+o** flag is used, the specified option is unset. When arguments are specified, they will cause positional parameters to be set or unset. Arguments, as specified by the *Option* variable, can be one of the following:

allexport
Same as the **-a** flag.

bgnice
Runs all background jobs at a lower priority. This is the default mode.

emacs
Enters an emacs-style inline editor for command entry.

errexit Same as the **-e** flag.

gmacs
Enters a gmacs-style inline editor for command entry.

ignoreeof
Does not exit the shell when it encounters an end-of-file character. To exit the shell, you must use the **exit** command, or press the Ctrl-D key sequence more than 11 times.

keyword
Same as the **-k** flag.

Note: This flag is for backward compatibility with the Bourne shell only. Its use is strongly discouraged.

markdirs
Appends a / to all directory names that are a result of file-name substitution.

monitor
Same as the **-m** flag.

noclobber
Prevents redirection from truncating existing files. When you specify this option, a vertical bar must follow the redirection symbol (>|) to truncate a file.

noexec
Same as the **-n** flag.

noglob
Same as the **-f** flag.

nolog Prevents function definitions in .profile and $ENV files from being saved in the history file.

nounset
Same as the **-u** flag.

privileged
Same as the **-p** flag.

trackall
Same as the **-h** flag.

verbose
Same as the **-v** flag.

vi Enters the insert mode of a vi-style inline editor for command entry. Entering escape character 033 puts the editor into the move mode. A return sends the line.

viraw Processes each character as it is typed in vi mode.

xtrace Same as the **-x** flag.

-p Disables processing of the **$HOME/.profile** file and uses the **/etc/suid _profile** file instead of the **ENV** file. This mode is enabled whenever the effective user ID (UID) or group ID (GID) is not equal to the real UID or GID. Turning off this option sets the effective UID or GID to the real UID and GID.

Note: The system does not support the **-p** option since the operating system does not support **setuid** shell scripts.

-s Sorts the positional parameters lexicographically.

-t Exits after reading and executing one command.

Note: This flag is for backward compatibility with the Bourne shell only. Its use is strongly discouraged.

-u
Treats unset parameters as errors when substituting.

-v Prints shell input lines as they are read.

-x Prints commands and their arguments as they are executed.

- Turns off the **-x** and **-v** flags and stops examining arguments for flags.

— Prevents any flags from being changed. This option is useful in setting the **$1** parameter to a value beginning with a **-**. If no arguments follow this flag, the positional parameters are not set.

Preceding any of the **set** command flags with a **+** rather than a **-** turns off the flag. You can use these flags when you invoke the shell. The current set of flags is found in the **$-** parameter. Unless you specify the **-A** flag, the remaining arguments are positional parameters and are assigned, in order, to `$1`, `$2`, ..., and so forth. If no arguments are given, the names and values of all named parameters are printed to standard output.

shift [*n*] Renames the positional parameters, beginning with `$n+1` ... through `$1` .... The default value of the *n* parameter is 1. The *n* parameter is any arithmetic expression that evaluates to a nonnegative number less than or equal to the $# parameter.

times Prints the accumulated user and system times for the shell and for processes run from the shell.

trap [*Command*] [*Signal*] ...	Runs the specified command when the shell receives the specified signal or signals. The *Command* parameter is read once when the trap is set and once when the trap is taken. The *Signal* parameter can be given as a number or as the name of the signal. Trap commands are executed in order of signal number. Any attempt to set a trap on a signal that was ignored on entry to the current shell is ineffective. If the command is a **-**, all traps are reset to their original values. If you omit the command and the first signal is a numeric signal number, then the **ksh** command resets the value of the *Signal* parameter or parameters to the original values. **Note:** If you omit the command and the first signal is a symbolic name, the signal is interpreted as a command. If the value of the *Signal* parameter is the **ERR** signal, the specified command is carried out whenever a command has a nonzero exit status. If the signal is **DEBUG**, then the specified command is carried out after each command. If the value of the *Signal* parameter is the **0** or **EXIT** signal and the **trap** command is executed inside the body of a function, the specified command is carried out after the function completes. If the *Signal* parameter is **0** or **EXIT** for a **trap** command set outside any function, the specified command is carried out on exit from the shell. The **trap** command with no arguments prints a list of commands associated with each signal number. For a complete list of *Signal* parameter values, used in the **trap** command without the **SIG** prefix, refer to the **sigaction**, **sigvec**, or **signal** subroutine in the *AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2*.
typeset [**+HLRZfilrtux** [*n*]] [*Name*[**=** *Value*]] ...	Sets attributes and values for shell parameters. When invoked inside a function, a new instance of the *Name* parameter is created. The parameter value and type are restored when the function completes. You can specify the following flags with the **typeset** command:

-H	Provides AIX-to-host-file mapping on non-AIX machines.
-L	Left-justifies and removes leading blanks from the *Value* parameter. If the *n* parameter has a nonzero value, it defines the width of the field; otherwise, it is determined by the width of the value of its first assignment. When the parameter is assigned, it is filled on the right with blanks or truncated, if necessary, to fit into the field. Leading zeros are removed if the **-Z** flag is also set. The **-R** flag is turned off.
-R	Right-justifies and fills with leading blanks. If the *n* parameter has a nonzero value, it defines the width of the field; otherwise, it is determined by the width of the value of its first assignment. The field remains filled with blanks or is truncated from the end if the parameter is reassigned. The **L** flag is turned off.
-Z	Right-justifies and fills with leading zeros if the first nonblank character is a digit and the **-L** flag has not been set. If the *n* parameter has a nonzero value, it defines the width of the field; otherwise, it is determined by the width of the value of its first assignment.
-f	Indicates that the names refer to function, rather than parameter, names. No assignments can be made and the only other valid flags are **-t**, **-u**, and **-x** . The **-t** flag turns on execution tracing for this function. The **-u** flag causes this function to be marked undefined. The **FPATH** variable is searched to find the function definition when the function is referenced. The **-x** flag allows the function definition to remain in effect across shell scripts that are not a separate invocation of the **ksh** command.
-i	Identifies the parameter as an integer, making arithmetic faster. If the *n* parameter has a nonzero value, it defines the output arithmetic base; otherwise, the first assignment determines the output base.
-l	Converts all uppercase characters to lowercase. The **-u** uppercase conversion flag is turned off.
-r	Marks the names specified by the *Name* parameter as read-only. These names cannot be changed by subsequent assignment.

-t	Tags the named parameters. Tags can be defined by the user and have no special meaning to the shell.
-u	Converts all lowercase characters to uppercase characters. The **-l** lowercase flag is turned off.
-x	Marks the name specified by the *Name* parameter for automatic export to the environment of subsequently executed commands.

Using a **+** rather than a **-** turns off the **typeset** command flags. If you do not specify *Name* parameters but do specify flags, a list of names (and optionally the values) of the parameters that have these flags set is printed. (Using a **+** rather than a **-** keeps the values from being printed.) If you do not specify any names or flags, the names and attributes of all parameters are printed.

unset [**-fv**] *Name* ...
Unsets the values and attributes of the parameters given by the list of names. If **-v** is specified, *Name* refers to a variable name, and the shell will unset it and remove it from the environment. Read-only variables cannot be unset. Unsetting the **ERRNO**, **LINENO**, **MAILCHECK**, **OPTARG**, **OPTIND**, **RANDOM**, **SECONDS**, **TMOUT**, and underscore (_) variables removes their special meanings even if they are subsequently assigned.

If the **-f** flag is set, then *Name* refers to a function name, and the shell will unset the function definition.

Regular Built-in Command Descriptions

The Korn Shell provides the following regular built-in commands:

```
alias          fg             print          ulimit
bg             getopts        pwd            umask
cd             jobs           read           unalias
command        kill           setgroups      wait
echo           let            test           whence
fc
```

alias [**-t**] [**-x**] [*AliasName*[**=** *String*]] ...
Creates or redefines alias definitions or writes existing alias definitions to standard output.

For more information, refer to the **alias** command in the *AIX 5L Version 5.2 Commands Reference*.

bg [*JobID*...]
Puts each specified job into the background. The current job is put in the background if a *JobID* parameter is not specified. Refer to "Job Control in the Korn Shell or POSIX Shell" on page 175 for more information about job control.

For more information about running jobs in the background, refer to the **bg** command in the *AIX 5L Version 5.2 Commands Reference*.

cd [*Argument*]
cd *Old New*
This command can be in either of two forms. In the first form, it changes the current directory to the one specified by the *Argument* parameter. If the value of the *Argument* parameter is -, the directory is changed to the previous directory. The **HOME** shell variable is the default value of the *Argument* parameter. The **PWD** variable is set to the current directory.

The **CDPATH** shell variable defines the search path for the directory containing the value of the *Argument* parameter. Alternative directory names are separated by a :. The default path is null, specifying the current directory. The current directory is specified by a null path name, which appears immediately after the equal sign or between the colon delimiters anywhere in the path list. If the specified argument begins with a /, the search path is not used. Otherwise, each directory in the path is searched for the argument.

The second form of the **cd** command substitutes the string specified by the *New* variable for the string specified by the *Old* variable in the current directory name, **PWD**, and tries to change to this new directory.

command [**-p**] *CommandName* [*Argument* ...] **command** [**-v** | **-V**] *CommandName*	Causes the shell to treat the specified command and arguments as a simple command, suppressing shell-function lookup. For more information, refer to the **command** command in the *AIX 5L Version 5.2 Commands Reference*.
echo [*String* ...]	Writes character strings to standard output. Refer to the **echo** command for usage and description. The **-n** flag is not supported.
fc [**-r**] [**-e** *Editor*] [*First* [*Last*]] **fc -l** [**-n**] [**-r**] [*First* [*Last*]] **fc -s** [*Old*= *New*] [*First*]	Displays the contents of your command history file or invokes an editor to modify and re-executes commands previously entered in the shell. For more information, refer to the **fc** command in the *AIX 5L Version 5.2 Commands Reference*.
fg [*JobID*]	Brings each job specified into the foreground. If you do not specify any jobs, the command brings the current job into the foreground. For more information about running jobs in the foreground, refer to the **fg** command in the *AIX 5L Version 5.2 Commands Reference*.
getopts *OptionString Name* [*Argument* ...]	Checks the *Argument* parameter for legal options. For more information, refer to the **getopts** command in the *AIX 5L Version 5.2 Commands Reference*.
jobs [**-l** | **-n** | **-p**] [*JobID* ...]	Displays the status of jobs started in the current shell environment. If no specific job is specified with the *JobID* parameter, status information for all active jobs is displayed. If a job termination is reported, the shell removes that job's process ID from the list of those known by the current shell environment. For more information, refer to the **jobs** command in the *AIX 5L Version 5.2 Commands Reference*.
kill [**-s** { *SignalName* | *SignalNumber* }] *ProcessID...* **kill** [*-SignalName* | *-SignalNumber*] *ProcessID...*	Sends a signal (by default, the **SIGTERM** signal) to a running process. This default action normally stops processes. If you want to stop a process, specify the process ID (PID) in the *ProcessID* variable. The shell reports the PID of each process that is running in the background (unless you start more than one process in a pipeline, in which case the shell reports the number of the last process). You can also use the **ps** command to find the process ID number of commands.
kill -l [*ExitStatus*]	Lists signal names. For more information, refer to the **kill** command in the *AIX 5L Version 5.2 Commands Reference*.
let *Expression* ...	Evaluates specified arithmetic expressions. The exit status is 0 if the value of the last expression is nonzero, and 1 otherwise. Refer to "Arithmetic Evaluation in the Korn Shell or POSIX Shell" on page 158 for more information.

print [**-Rnprsu** [*n*]] [*Argument* ...]	Prints shell output. If you do not specify any flags, or if the hyphen (**-**) or double hyphen (**—**) flags are specified, the arguments are printed to standard output as described by the **echo** command. The flags do the following: **-R** Prints in raw mode (the escape conventions of the **echo** command are ignored). The **-R** Flag prints all subsequent arguments and flags other than **-n**. **-n** Prevents a new-line character from being added to the output. **-p** Writes the arguments to the pipe of the process run with **|&** instead of to standard output. **-r** Prints in raw mode. The escape conventions of the **echo** command are ignored. **-s** Writes the arguments to the history file instead of to standard output. **-u** Specifies a one-digit file descriptor unit number, *n*, on which the output is placed. The default is 1.
pwd	Equivalent to **print -r - $PWD**. **Note:** The internal Korn shell **pwd** command does not support symbolic links.
read [**-prsu** [**n**]] [*Name?Prompt*] [*Name...*]	Takes shell input. One line is read and broken up into fields, using the characters in the **IFS** variable as separators. For more information, refer to the **read** command in the *AIX 5L Version 5.2 Commands Reference*.
setgroups	Executes the **/usr/bin/setgroups** command, which runs as a separate shell. See the **setgroups** command for information on this command. There is one difference, however. The **setgroups** built-in command invokes a subshell, but the **setgroups** command replaces the currently executing shell. Because the built-in command is supported only for compatibility, it is recommended that scripts use the absolute path name **/usr/bin/setgroups** rather than the shell built-in command.
test	Same as [*expression*]. See “Conditional Expressions for the Korn Shell or POSIX Shell” on page 174 for usage and description.

ulimit [**-HSacdfmst**] [*Limit*]

Sets or displays user-process resource limits as defined in the **/etc/security/limits** file. This file contains the following default limits:

```
fsize = 2097151
core = 2048
cpu = 3600
data = 131072
rss = 65536
stack = 8192
```

These values are used as default settings when a user is added to the system. The values are set with the **mkuser** command when the user is added to the system, or changed with the **chuser** command.

Limits are categorized as either soft or hard. Users might change their soft limits, up to the maximum set by the hard limits, with the **ulimit** command. You must have root user authority to change resource hard limits.

Many systems do not contain one or more of these limits. The limit for a specified resource is set when the *Limit* parameter is specified. The value of the *Limit* parameter can be a number in the unit specified with each resource, or the value `unlimited`. You can specify the following **ulimit** command flags:

-H	Specifies that the hard limit for the given resource is set. If you have root user authority, you can increase the hard limit. Any user can decrease it.
-S	Specifies that the soft limit for the given resource is set. A soft limit can be increased up to the value of the hard limit. If neither the **-H** or **-S** options are specified, the limit applies to both.
-a	Lists all of the current resource limits.
-c	Specifies the number of 512-byte blocks on the size of core dumps.
-d	Specifies the size, in KB, of the data area.
-f	Specifies the number of 512-byte blocks for files written by child processes (files of any size can be read).
-m	Specifies the number of KB for the size of physical memory.
-n	Specifies the limit on the number of file descriptors a process might have open.
-s	Specifies the number of KB for the size of the stack area.
-t	Specifies the number of seconds to be used by each process.

The current resource limit is printed when you omit the *Limit* variable. The soft limit is printed unless you specify the **-H** flag. When you specify more than one resource, the limit name and unit is printed before the value. If no option is given, the **-f** flag is assumed. When you change the value, set both hard and soft limits to *Limit* unless you specify **-H** or **-S**.

For more information about user and system resource limits, refer to the **getrlimit**, **setrlimit**, or **vlimit** subroutine in the *AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1*.

umask [**-S**] [*Mask*]

Determines file permissions. This value, along with the permissions of the creating process, determines a file's permissions when the file is created. The default is 022. If the Mask parameter is not specified, the **umask** command displays to standard output the file-mode creation mask of the current shell environment.

For more information about file permissions, refer to the **umask** command in the *AIX 5L Version 5.2 Commands Reference*.

unalias { **-a** | *AliasName...* }

Removes the definition for each alias name specified, or removes all alias definitions if the **-a** flag is used. Alias definitions are removed from the current shell environment.

For more information, refer to the **unalias** command in the *AIX 5L Version 5.2 Commands Reference*.

wait [*ProcessID...*]	Waits for the specified job and terminates. If you do not specify a job, the command waits for all currently active child processes. The exit status from this command is that of the process for which it waits. For more information, refer to the **wait** command in the *AIX 5L Version 5.2 Commands Reference*.
whence [**-pv**] *Name* ...	Indicates, for each name specified, how it would be interpreted if used as a command name. When used without either flag, **whence** will display the absolute path name, if any, that corresponds to each name. **-p** Does a path search for the specified name or names even if these are aliases, functions, or reserved words. **-v** Produces a more verbose report that specifies which type each name is.

List of Korn Shell or POSIX Shell Built-in Commands

Special Built-in Commands

: (colon)	Expands only arguments.
. (dot)	Reads a specified file and then executes the commands.
break	Exits from the enclosing **for**, **while**, **until**, or **select** loop, if one exists.
continue	Resumes the next iteration of the enclosing **for**, **while**, **until**, or **select** loop.
eval	Reads the arguments as input to the shell and executes the resulting command or commands.
exec	Executes the command specified by the *Argument* parameter, instead of this shell, without creating a new process.
exit	Exits the shell whose exit status is specified by the *n* parameter.
export	Marks names for automatic export to the environment of subsequently executed commands.
newgrp	Equivalent to the **exec /usr/bin/newgrp** [*Group ...*] command.
readonly	Marks the specified names read-only.
return	Causes a shell to return to the invoking script.
set	Unless options or arguments are specified, writes the names and values of all shell variables in the collation sequence of the current locale.
shift	Renames positional parameters.
times	Prints the accumulated user and system times for both the shell and the processes run from the shell.
trap	Runs a specified command when the shell receives a specified signal or signals.
typeset	Sets attributes and values for shell parameters.
unset	Unsets the values and attributes of the specified parameters.

Regular Built-in Commands

alias	Prints a list of aliases to standard output.
bg	Puts specified jobs in the background.
cd	Changes the current directory to the specified directory or substitutes the current string with the specified string.
echo	Writes character strings to standard output.
fc	Selects a range of commands from the last HISTSIZE variable command typed at the terminal. Re-executes the specified command after old-to-new substitution is performed.
fg	Brings the specified job to the foreground.
getopts	Checks the *Argument* parameter for legal options.
jobs	Lists information for the specified jobs.
kill	Sends the **TERM** (terminate) signal to specified jobs or processes.
let	Evaluates specified arithmetic expressions.
print	Prints shell output.

pwd	Equivalent to the **print -r -$PWD** command**.**
read	Takes shell input.
ulimit	Sets or displays user process resource limits as defined in the **/etc/security/limits** file.
umask	Determines file permissions.
unalias	Removes the parameters in the list of names from the alias list.
wait	Waits for the specified job and terminates.
whence	Indicates how each specified name would be interpreted if used as a command name.

For more information, see "Korn Shell or POSIX Shell Built-In Commands" on page 163.

Conditional Expressions for the Korn Shell or POSIX Shell

A conditional expression is used with the **[[** compound command to test attributes of files and to compare strings. Word splitting and file name substitution are not performed on words appearing between **[[** and **]]**. Each expression is constructed from one or more of the following unary or binary expressions:

-a *File*	True, if the specified file is a symbolic link that points to another file that does exist.
-b *File*	True, if the specified file exists and is a block special file.
-c *File*	True, if the specified file exists and is a character special file.
-d *File*	True, if the specified file exists and is a directory.
-e *File*	True, if the specified file exists.
-f *File*	True, if the specified file exists and is an ordinary file.
-g *File*	True, if the specified file exists and its **setgid** bit is set.
-h *File*	True, if the specified file exists and is a symbolic link.
-k *File*	True, if the specified file exists and its sticky bit is set.
-n *String*	True, if the length of the specified string is nonzero.
-o *Option*	True, if the specified option is on.
-p *File*	True, if the specified file exists and is a FIFO special file or a pipe.
-r *File*	True, if the specified file exists and is readable by the current process.
-s *File*	True, if the specified file exists and has a size greater than 0.
-t *FileDescriptor*	True, if specified file descriptor number is open and associated with a terminal device.
-u *File*	True, if the specified file exists and its **setuid** bit is set.
-w *File*	True, if the specified file exists and the write bit is on. However, the file will not be writable on a read-only file system even if this test indicates true.
-x *File*	True, if the specified file exists and the **execute** flag is on. If the specified file exists and is a directory, then the current process has permission to search in the directory.
-z *String*	True, if length of the specified string is 0.
-L *File*	True, if the specified file exists and is a symbolic link.
-O *File*	True, if the specified file exists and is owned by the effective user ID of this process.
-G *File*	True, if the specified file exists and its group matches the effective group ID of this process.
-S *File*	True, if the specified file exists and is a socket.
File1 **-nt** *File2*	True, if *File1* exists and is newer than *File2*.
File1 **-ot** *File2*	True, if *File1* exists and is older than *File2*.
File1 **-ef** *File2*	True, if *File1* and *File2* exist and refer to the same file.
String1 **=** *String2*	True, if *String1* is equal to *String2*.
String1 **!=** *String2*	True, if *String1* is not equal to *String2*.
String **=** *Pattern*	True, if the specified string matches the specified pattern.
String **!=** *Pattern*	True, if the specified string does not match the specified pattern.
String1 **<** *String2*	True, if *String1* comes before *String2* based on the ASCII value of their characters.
String1 **>** *String2*	True, if *String1* comes after *String2* based on the ASCII value of their characters.
Expression1 **-eq** *Expression2*	True, if *Expression1* is equal to *Expression2*.
Expression1 **-ne** *Expression2*	True, if *Expression1* is not equal to *Expression2*.

Expression1 **-lt** *Expression2*	True, if *Expression1* is less than *Expression2*.
Expression1 **-gt** *Expression2*	True, if *Expression1* is greater than *Expression2*.
Expression1 **-le** *Expression2*	True, if *Expression1* is less than or equal to *Expression2*.
Expression1 **-ge** *Expression2*	True, if *Expression1* is greater than or equal to *Expression2*.

Note: In each of the previous expressions, if the *File* variable is similar to `/dev/fd/n`, where `n` is an integer, then the test is applied to the open file whose descriptor number is *n*.

You can construct a compound expression from these primitives, or smaller parts, by using any of the following expressions, listed in decreasing order of precedence:

(*Expression***)**	True, if the specified expression is true. Used to group expressions.
! *Expression*	True, if the specified expression is false.
Expression1 **&&** *Expression2*	True, if *Expression1* and *Expression2* are both true.
Expression1 || *Expression2*	True, if either *Expression1* or *Expression2* is true.

Job Control in the Korn Shell or POSIX Shell

The Korn shell, or POSIX shell, provides a facility to control command sequences, or *jobs*. When you execute the **set -m** special command, the Korn shell associates a job with each pipeline. It keeps a table of current jobs, printed by the **jobs** command, and assigns them small integer numbers.

When a job is started in the background with an **&** , the shell prints a line that looks like the following:

```
[1] 1234
```

This output indicates that the job, which was started in the background, was job number 1. It also shows that the job had one (top-level) process with a process ID of `1234`.

If you are running a job and want to do something else, use the Ctrl-Z key sequence. This key sequence sends a **STOP** signal to the current job. The shell normally indicates that the job has been stopped and then displays a shell prompt. You can then manipulate the state of this job (putting it in the background with the **bg** command), run other commands, and then eventually return the job to the foreground with the **fg** command. The Ctrl-Z key sequence takes effect immediately, and is like an interrupt in that the shell discards pending output and unread input when you type the sequence.

A job being run in the background stops if it tries to read from the terminal. Background jobs are normally allowed to produce output. You can disable this option by issuing the **stty tostop** command. If you set this terminal option, then background jobs stop when they try to produce output or read input.

You can refer to jobs in the Korn shell in several ways. A job is referenced by the process ID of any of its processes, or in one of the following ways:

%*Number*	Specifies the job with the given number
%*String*	Specifies any job whose command line begins with the *String* variable
%?*String*	Specifies any job whose command line contains the *String* variable
%%	Specifies the current job
%+	Equivalent to **%%**
%-	Specifies the previous job

This shell immediately recognizes changes in the process state. It normally informs you whenever a job becomes blocked so that no further progress is possible. The shell does this just before it prints a prompt so that it does not otherwise disturb your work.

When the monitor mode is on, each completed background job triggers traps set for the **CHLD** signal.

If you try to leave the shell (either by typing `exit` or using the Ctrl-D key sequence) while jobs are stopped or running, the system warns you with the message `There are stopped (running) jobs.` Use the **jobs** command to see which jobs are affected. If you immediately try to exit again, the shell terminates the stopped and running jobs without warning.

Signal Handling

The **SIGINT** and **SIGQUIT** signals for an invoked command are ignored if the command is followed by **&** and the job **monitor** option is not active. Otherwise, signals have the values that the shell inherits from its parent.

When a signal for which a trap has been set is received while the shell is waiting for the completion of a foreground command, the trap associated with that signal will not be executed until after the foreground command has completed. Therefore, a trap on a **CHILD** signal is not performed until the foreground job terminates.

Inline Editing in the Korn Shell or POSIX Shell

Normally, you type each command line from a terminal device and follow it by a new-line character (RETURN or LINE FEED). When you activate the emacs, gmacs, or vi inline editing option, you can edit the command line.

The following commands enter edit modes:

set -o emacs	Enters emacs editing mode and initiates an emacs-style inline editor. For more information, see "emacs Editing Mode" on page 177.
set -o gmacs	Enters emacs editing mode and initiates a gmacs-style inline editor. For more information, see "emacs Editing Mode" on page 177.
set -o vi	Enters vi editing mode and initiates a vi-style inline editor. For more information, see "vi Editing Mode" on page 178.

An editing option is automatically selected each time the **VISUAL** or **EDITOR** variable is assigned a value that ends in any of these option names.

> **Note:** To use the editing features, your terminal must accept RETURN as a carriage return without line feed. A space must overwrite the current character on the screen.

Each editing mode opens a window at the current line. The window width is the value of the **COLUMNS** variable if it is defined; otherwise, the width is 80 character spaces. If the line is longer than the window width minus two, the system notifies you by displaying a mark at the end of the window. As the cursor moves and reaches the window boundaries, the window is centered about the cursor. The marks displayed are as follows:

>	Indicates that the line extends on the right side of the window.
<	Indicates that the line extends on the left side of the window.
*****	Indicates that the line extends on both sides of the window.

The search commands in each edit mode provide access to the Korn shell history file. Only strings are matched. If the leading character in the string is a ^, the match must begin at the first character in the line.

emacs Editing Mode

The emacs editing mode is entered when you enable either the **emacs** or **gmacs** option. The only difference between these two modes is the way each handles the Ctrl-T edit command. To edit, move the cursor to the point needing correction and insert or delete characters or words, as needed. All of the editing commands are control characters or escape sequences.

Edit commands operate from any place on a line (not only at the beginning). Do not press the Enter key or line-feed (Down Arrow) key after edit commands, except as noted.

Ctrl-F	Moves the cursor forward (right) one character.
Esc-F	Moves the cursor forward one word (a string of characters consisting of only letters, digits, and underscores).
Ctrl-B	Moves the cursor backward (left) one character.
Esc-B	Moves the cursor backward one word.
Ctrl-A	Moves the cursor to the beginning of the line.
Ctrl-E	Moves the cursor to the end of the line.
Ctrl-] *c*	Moves the cursor forward on the current line to the indicated character.
Esc-Ctrl-] *c*	Moves the cursor backward on the current line to the indicated character.
Ctrl-X Ctrl-X	Interchanges the cursor and the mark.
ERASE	Deletes the previous character. (User-defined erase character as defined by the **stty** command, usually the Ctrl-H key sequence.)
Ctrl-D	Deletes the current character.
Esc-D	Deletes the current word.
Esc-Backspace	Deletes the previous word.
Esc-H	Deletes the previous word.
Esc-Delete	Deletes the previous word. If your interrupt character is the Delete key, this command does not work.
Ctrl-T	Transposes the current character with the next character in emacs mode. Transposes the two previous characters in gmacs mode.
Ctrl-C	Capitalizes the current character.
Esc-C	Capitalizes the current word.
Esc-L	Changes the current word to lowercase.
Ctrl-K	Deletes from the cursor to the end of the line. If preceded by a numeric parameter whose value is less than the current cursor position, this editing command deletes from the given position up to the cursor. If preceded by a numeric parameter whose value is greater than the current cursor position, this editing command deletes from the cursor up to the given cursor position.
Ctrl-W	Deletes from the cursor to the mark.
Esc-P	Pushes the region from the cursor to the mark on the stack.
KILL	User-defined kill character as defined by the **stty** command, usually the Ctrl-G key sequence or an **@**. Kills the entire current line. If two kill characters are entered in succession, all subsequent kill characters cause a line feed (useful when using paper terminals).
Ctrl-Y	Restores the last item removed from the line. (Yanks the item back to the line.)
Ctrl-L	Line feeds and prints the current line.
Ctrl-@	(Null character) Sets a mark.
Esc-space	Sets a mark.
Ctrl-J	(New line) Executes the current line.
Ctrl-M	(Return) Executes the current line.
EOF	Processes the end-of-file character, normally the Ctrl-D key sequence, as an end-of-file only if the current line is null.

Ctrl-P	Fetches the previous command. Each time the Ctrl-P key sequence is entered, the previous command back in time is accessed. Moves back one line when not on the first line of a multiple-line command.
Esc-<	Fetches the least recent (oldest) history line.
Esc->	Fetches the most recent (youngest) history line.
Ctrl-N	Fetches the next command line. Each time the Ctrl-N key sequence is entered, the next command line forward in time is accessed.
Ctrl-R *String*	Reverses search history for a previous command line containing the string specified by the *String* parameter. If a value of 0 is given, the search is forward. The specified string is terminated by an Enter or new-line character. If the string is preceded by a ^, the matched line must begin with the *String* parameter. If the *String* parameter is omitted, then the next command line containing the most recent *String* parameter is accessed. In this case, a value of 0 reverses the direction of the search.
Ctrl-O	(Operate) Executes the current line and fetches the next line relative to the current line from the history file.
Esc *Digits*	(Escape) Defines the numeric parameter. The digits are taken as a parameter to the next command. The commands that accept a parameter are **Ctrl-F**, **Ctrl-B**, **ERASE**, **Ctrl-C**, **Ctrl-D**, **Ctrl-K**, **Ctrl-R**, **Ctrl-P**, **Ctrl-N**, **Ctrl-]**, **Esc-.**, **Esc-Ctrl-]**, **Esc-_**, **Esc-B**, **Esc-C**, **Esc-D**, **Esc-F**, **Esc-H**, **Esc-L**, and **Esc-Ctrl-H**.
Esc *Letter*	(Soft-key) Searches the alias list for an alias named *_Letter.* If an alias of this name is defined, its value is placed into the input queue. The *Letter* parameter must not specify one of the escape functions.
Esc-[*Letter*	(Soft-key) Searches the alias list for an alias named double underscore Letter (*__Letter*). If an alias of this name is defined, its value is placed into the input queue. This command can be used to program function keys on many terminals.
Esc-.	Inserts on the line the last word of the previous command. If preceded by a numeric parameter, the value of this parameter determines which word to insert rather than the last word.
Esc-_	Same as the Esc-. key sequence.
Esc-*	Attempts file-name substitution on the current word. An asterisk is appended if the word does not match any file or contain any special pattern characters.
Esc-Esc	File-name completion. Replaces the current word with the longest common prefix of all file names that match the current word with an asterisk appended. If the match is unique, a **/** is appended if the file is a directory and a space is appended if the file is not a directory.
Esc-=	Lists the files that match the current word pattern as if an asterisk were appended.
Ctrl-U	Multiplies the parameter of the next command by 4.
\	Escapes the next character. Editing characters and the ERASE, KILL and INTERRUPT (normally the Delete key) characters can be entered in a command line or in a search string if preceded by a \\. The backslash removes the next character's editing features, if any.
Ctrl-V	Displays the version of the shell.
Esc-#	Inserts a # at the beginning of the line and then executes the line. This causes a comment to be inserted in the history file.

vi Editing Mode

The vi editing mode has two typing modes. When you enter a command, you are in Input mode. To edit, you must enter the Control mode by pressing the Esc key.

Most control commands accept an optional repeat *Count* parameter prior to the command. When in vi mode on most systems, canonical processing is initially enabled. The command is echoed again if one or more of the following are true:

- The speed is 1200 baud or greater.
- The command contains any control characters.
- Less than one second has elapsed since the prompt was printed.

The Esc character terminates canonical processing for the remainder of the command, and you can then modify the command line. This scheme has the advantages of canonical processing with the type-ahead echoing of raw mode. If the **viraw** option is also set, canonical processing is always disabled. This mode is implicit for systems that do not support two alternate end-of-line delimiters and might be helpful for certain terminals.

Available vi edit commands are grouped into catagories. The categories are as follows:

- "Input Edit Commands"
- "Motion Edit Commands"
- "Search Edit Commands"
- "Text-Modification Edit Commands" on page 180
- "Miscellaneous Edit Commands" on page 181

Input Edit Commands

Note: By default, the editor is in input mode.

ERASE	(User-defined erase character as defined by the **stty** command, usually Ctrl-H or #.) Deletes the previous character.
Ctrl-W	Deletes the previous blank separated word.
Ctrl-D	Terminates the shell.
Ctrl-V	Escapes the next character. Editing characters, such as the ERASE or KILL characters, can be entered in a command line or in a search string if preceded by a Ctrl-V key sequence. The Ctrl-V key sequence removes the next character's editing features (if any).
\	Escapes the next ERASE or KILL character.

Motion Edit Commands

Motion edit commands move the cursor as follows:

[*Count*]**l**	Moves the cursor forward (right) one character.
[*Count*]**w**	Moves the cursor forward one alphanumeric word.
[*Count*]**W**	Moves the cursor to the beginning of the next word that follows a blank.
[*Count*]**e**	Moves the cursor to the end of the current word.
[*Count*]**E**	Moves the cursor to the end of the current blank-separated word.
[*Count*]**h**	Moves the cursor backward (left) one character.
[*Count*]**b**	Moves the cursor backward one word.
[*Count*]**B**	Moves the cursor to the previous blank-separated word.
[*Count*]**l**	Moves the cursor to the column specified by the *Count* parameter.
[*Count*]**f***c*	Finds the next character *c* in the current line.
[*Count*]**F***c*	Finds the previous character *c* in the current line.
[*Count*]**t***c*	Equivalent to **f** followed by **h**.
[*Count*]**T***c*	Equivalent to **F** followed by **l**.
[*Count*]**;**	Repeats for the number of times specified by the *Count* parameter the last single-character find command: **f**, **F**, **t**, or **T**.
[*Count*]**,**	Reverses the last single-character find command the number of times specified by the *Count* parameter.
0	Moves the cursor to the start of a line.
^	Moves the cursor to the first nonblank character in a line.
$	Moves the cursor to the end of a line.

Search Edit Commands

Search edit commands access your command history, as follows:

[*Count*]**k**	Fetches the previous command.
[*Count*]**-**	Equivalent to the **k** command.
[*Count*]**j**	Fetches the next command. Each time the **j** command is entered, the next command is accessed.

[*Count*]**+**	Equivalent to the **j** command.
[*Count*]**G**	Fetches the command whose number is specified by the *Count* parameter. The default is the least recent history command.
/*String*	Searches backward through history for a previous command containing the specified string. The string is terminated by a RETURN or newline character. If the specified string is preceded by a ^, the matched line must begin with the *String* parameter. If the value of the *String* parameter is null, the previous string is used.
?*String*	Same as **/***String* except that the search is in the forward direction.
n	Searches for the next match of the last pattern to **/***String* or **?** commands.
N	Searches for the next match of the last pattern to **/***String* or **?** commands, but in the opposite direction. Searches history for the string entered by the previous **/***String* command.

Text-Modification Edit Commands

Text-modification edit commands modify the line as follows:

a	Enters the input mode and enters text after the current character.
A	Appends text to the end of the line. Equivalent to the **$a** command.
[*Count*]**c***Motion*	
c[*Count*]*Motion*	Deletes the current character through the character to which the *Motion* parameter specifies to move the cursor, and enters input mode. If the value of the *Motion* parameter is **c**, the entire line is deleted and the input mode is entered.
C	Deletes the current character through the end of the line and enters input mode. Equivalent to the **c$** command.
S	Equivalent to the **cc** command.
D	Deletes the current character through the end of line. Equivalent to the **d$** command.

[*Count*]**d***Motion*

d[*Count*]*Motion*	Deletes the current character up to and including the character specified by the *Motion* parameter. If *Motion* is **d**, the entire line is deleted.
i	Enters the input mode and inserts text before the current character.
I	Inserts text before the beginning of the line. Equivalent to the **0i** command.
[*Count*]**P**	Places the previous text modification before the cursor.
[*Count*]**p**	Places the previous text modification after the cursor.
R	Enters the input mode and types over the characters on the screen.
[*Count*]**r***c*	Replaces the number of characters specified by the *Count* parameter, starting at the current cursor position, with the characters specified by the *c* parameter. This command also advances the cursor after the characters are replaced.
[*Count*]**x**	Deletes the current character.
[*Count*]**X**	Deletes the preceding character.
[*Count*]**.**	Repeats the previous text-modification command.
[*Count*]~	Inverts the case of the number of characters specified by the *Count* parameter, starting at the current cursor position, and advances the cursor.
[*Count*]_	Appends the word specified by the *Count* parameter of the previous command and enters input mode. The last word is used if the *Count* parameter is omitted.
*	Appends an * to the current word and attempts file-name substitution. If no match is found, it rings the bell. Otherwise, the word is replaced by the matching pattern and input mode is entered.
\	File name completion. Replaces the current word with the longest common prefix of all file names matching the current word with an asterisk appended. If the match is unique, a **/** is appended if the file is a directory. A space is appended if the file is not a directory.

Miscellaneous Edit Commands

The most commonly used edit commands include the following:

[*Count*]**y***Motion*	
y[*Count*]*Motion*	Yanks the current character up to and including the character marked by the cursor position specified by the *Motion* parameter and puts all of these characters into the delete buffer. The text and cursor are unchanged.
Y	Yanks from the current position to the end of the line. Equivalent to the **y$** command.
u	Undoes the last text-modifying command.
U	Undoes all the text-modifying commands performed on the line.
[*Count*]**v**	Returns the command `fc -e ${VISUAL:-${EDITOR:-vi}} Count` in the input buffer. If the *Count* parameter is omitted, then the current line is used.
Ctrl-L	Line feeds and prints the current line. This command is effective only in control mode.
Ctrl-J	(New line) Executes the current line, regardless of the mode.
Ctrl-M	(Return) Executes the current line, regardless of the mode.
#	Sends the line after inserting a **#** in front of the line. Useful if you want to insert the current line in the history without executing it. If the command line contains a pipe or semicolon or newline character, then additional **#**s will be inserted in front of each of these symbols. To delete all pound signs, retrieve the command line from history and enter another **#**.
=	Lists the file names that match the current word as if an asterisk were appended to it.
@*Letter*	Searches the alias list for an alias named _*Letter*. If an alias of this name is defined, its value is placed into the input queue for processing.

Enhanced Korn Shell (ksh93)

In addition to the default system Korn shell (**/usr/bin/ksh**), AIX provides an enhanced version available as **/usr/bin/ksh93**. This enhanced version is upwardly compatible with the current default version, and includes a few additional features that are not available in **/usr/bin/ksh**.

The following features are available in **/usr/bin/ksh93**:

Arithmetic Enhancements	You can use libm functions (math functions typically found in the C programming language), within arithmetic expressions, such as `$ value=$((sqrt(9)))`. More arithmetic operators are available, including the unary +, ++, **--**, and the **?:** construct (for example, ″`x ? y : z`″), as well as the **,** (comma) operator. Arithmetic bases are supported up to base 64. Floating point arithmetic is also supported. ″`typeset -E`″ (exponential) can be used to specify the number of significant digits and ″`typeset -F`″ (float) can be used to specify the number of decimal places for an arithmetic variable. The `SECONDS` variable now displays to the nearest hundredth of a second, rather than to the nearest second.

Compound Variables	Compound variables are supported. A compound variable allows a user to specify multiple values within a single variable name. The values are each assigned with a subscript variable, separated from the parent variable with a **.** (period). For example:
	`$ myvar=( x=1 y=2 )`
	`$ print "${myvar.x}"`
	`1`

Compound Assignments	Compound assignments are supported when initializing arrays, both for indexed arrays and associative arrays. The assignment values are placed in parentheses, as shown in the following example:
	`$ numbers=( zero one two three )`
	`$ print ${numbers[0]} ${numbers[3]}`
	`zero three`

Associative Arrays	An associative array is an array with a string as an index.
	
	The **typeset** command used with the **-A** flag allows you to specify associative arrays within ksh93. For example:
	`$ typeset -A teammates`
	`$ teammates=( [john]=smith [mary]=jones )`
	`$ print ${teammates[mary]}`
	`jones`

Variable Name References	The **typeset** command used with the **-n** flag allows you to assign one variable name as a reference to another. In this way, modifying the value of a variable will in turn modify the value of the variable that is referenced. For example:
	`$ greeting="hello"`
	`$ typeset -n welcome=greeting     # establishes the reference`
	`$ welcome="hi there"     # overrides previous value`
	`$ print $greeting`
	`hi there`

Parameter Expansions	The following parameter-expansion constructs are available: • `${!varname}` is the name of the variable itself. • `${!`*varname*`[@]}` names the indexes for the *varname* array. • `${`*param*`:`*offset*`}` is a substring of *param*, starting at *offset*. • `${param:offset:num}` is a substring of *param*, starting at *offset*, for *num* number of characters. • `${@:`*offset*`}` indicates all positional parameters starting at *offset*. • `${@:`*offset*`:`*num*`}` indicates *num* positional parameters starting at *offset*. • `${`*param*`/`*pattern*`/`*repl*`}` evaluates to *param*, with the first occurrence of *pattern* replaced by *repl*. • `${`*param*`//`*pattern*`/`*repl*`}` evaluates to *param*, with every occurrence of *pattern* replaced by *repl*. • `${`*param*`/#`*pattern*`/`*repl*`}` if *param* begins with *pattern*, then *param* is replaced by *repl*. • `${`*param*`/%`*pattern*`/`*repl*`}` if *param* ends with *pattern*, then *param* is replaced by *repl*.

Discipline Functions	A discipline function is a function that is associated with a specific variable. This allows you to define and call a function every time that variable is referenced, set, or unset. These functions take the form of *varname.function*, where *varname* is the name of the variable and *function* is the discipline function. The predefined discipline functions are **get**, **set**, and **unset**.
	• The *varname***.get** function is invoked every time *varname* is referenced. If the special variable **.sh.value** is set within this function, then the value of *varname* is changed to this value. A simple example is the time of day:
	`$ function time.get`
	`> {`
	`>     .sh.value=$(date +%r)`
	`> }`
	`$ print $time`
	`09:15:58 AM`
	`$ print $time    # it will change in a few seconds`
	`09:16:04 AM`
	• The *varname***.set** function is invoked every time *varname* is set. The **.sh.value** variable is given the value that was assigned. The value assigned to *varname* is the value of **.sh.value** when the function completes. For example:
	`$ function adder.set`
	`> {`
	`>   let .sh.value="`
	`$ {.sh.value} + 1"`
	`> }`
	`$ adder=0`
	`$ echo $adder`
	`1`
	`$ adder=$adder`
	`$ echo $adder`
	`2`
	• The *varname***.unset** function is executed every time *varname* is unset. The variable is not actually unset unless it is unset within the function itself; otherwise it retains its value.
	Within all discipline functions, the special variable **.sh.name** is set to the name of the variable, while **.sh.subscript** is set to the value of the variables subscript, if applicable.

Function Environments	Functions declared with the `function` *`myfunc`* format are executed in a separate function environment. Functions declared as *`myfunc`*`()` execute with the same environment as the parent shell.
Variables	Variables beginning with `.sh.` are reserved by the shell and have special meaning. See the description of Discipline Functions in this table for an explanation of **.sh.name**, **.sh.value**, and **.sh.subscript**. Also available is **.sh.version**, which represents the version of the shell.
Command Return Values	Return values of commands are as follows: • If the command to be executed is not found, the return value is set to 127. • If the command to be executed is found, but not executable, the return value is 126. • If the command is executed, but is terminated by a signal, the return value is 256 plus the signal number.
PATH Search Rules	Special built-in commands are searched for first, followed by all functions (including those in FPATH directories), followed by other built-ins.
Shell History	The **hist** command allows you to display and edit the shells command history. In the ksh shell, the **fc** command was used. The **fc** command is an alias to **hist**. Variables are HISTCMD, which increments once for each command executed in the shells current history, and HISTEDIT, which specifies which editor to use when using the **hist** command.

Built-In Commands

The enhanced Korn shell contains the following built-in commands:

- The **builtin** command lists all available built-in commands.
- The **printf** command works in a similar manner as the printf() C library routine. Refer to the **printf** command.
- The **disown** blocks the shell from sending a SIGHUP to the specified command.
- The **getconf** command works in the same way as the stand-alone command **/usr/bin/getconf**. Refer to the **getconf** command.
- The **read** built-in command has the following flags:
  - **read -d {***char***}** allows you to specify a character delimiter instead of the default newline.
  - **read -t {***seconds***}** allows you to specify a time limit in seconds after which the **read** command will time out. If **read** times out, it will return FALSE.
- The **exec** built-in command has the following flags:
  - **exec -a {***name***} {***cmd***}** specifies that argument 0 of *cmd* be replaced with *name*.
  - **exec -c {***cmd***}** tells **exec** to clear the environment before executing *cmd*.
- The **kill** built-in command has the following flags:
  - **kill -n {***signum***}** is used for specifying a signal number to send to a process, while **kill -s {***signame***}** is used to specify a signal name.
  - **kill -l**, with no arguments, lists all signal names but not their numbers.
- The **whence** built-in command has the following flags:
  - The **-a** flag displays all matches, not only the first one found.
  - The **-f** flag tells **whence** not to search for any functions.
- An escape character sequence is used for use by the **print** and **echo** commands. The Esc (Escape) key can be represented by the sequence `\E`.
- All regular built-in commands recognize the **-?** flag, which shows the syntax for the specified command.

Bourne Shell

The Bourne shell is an interactive command interpreter and command programming language. The **bsh** command runs the Bourne shell.

The Bourne shell can be run either as a login shell or as a subshell under the login shell. Only the **login** command can call the Bourne shell as a login shell. It does this by using a special form of the **bsh** command name: `-bsh`. When called with an initial hyphen (-), the shell first reads and runs commands found in the system **/etc/profile** file and your **$HOME/.profile**, if one exists. The**/etc/profile** file sets variables needed by all users. Finally, the shell is ready to read commands from your standard input.

If the *File* [*Parameter*] parameter is specified when the Bourne shell is started, the shell runs the script file identified by the *File* parameter, including any parameters specified. The script file specified must have read permission; any **setuid** and **setgid** settings are ignored. The shell then reads the commands. If either the **-c** or **-s** flag is used, do not specify a script.

Bourne Shell Environment

All variables (with their associated values) known to a command at the beginning of its execution constitute its *environment*. This environment includes variables that a command inherits from its parent process and variables specified as keyword parameters on the command line that calls the command.

The shell passes to its child processes the variables named as arguments to the built-in **export** command. This command places the named variables in the environments of both the shell and all its future child processes.

Keyword parameters are variable-value pairs that appear in the form of assignments, normally before the procedure name on a command line (but see also the flag for the **set** command). These variables are placed in the environment of the procedure being called.

For example, consider the following procedure, which displays the values of two variables (saved in a command file named `key_command`):

```
# key_command
echo $a $b
```

The following command lines produce the output shown:

```
Input                                 Output
a=key1   b=key2  key_command          key1 key2
a=tom    b=john  key_command          tom john
```

A procedure's keyword parameters are not included in the parameter count stored in $#.

A procedure can access the values of any variables in its environment. If it changes any of these values, however, the changes are not reflected in the shell environment. The changes are local to the procedure in question. To place the changes in the environment that the procedure passes to its child processes, you must export the new values within that procedure.

To obtain a list of variables that are exportable from the current shell, type:

```
export
```

Press Enter.

To obtain a list of read-only variables from the current shell, type:

```
readonly
```

Press Enter.

To obtain a list of variable-value pairs in the current environment, type:

```
env
```

Press Enter.

For more information about user environments, see "/etc/environment File" on page 130

Restricted Shell

The restricted shell is used to set up login names and execution environments whose capabilities are more controlled than those of the regular Bourne shell. The **Rsh** or **bsh -r** command opens the restricted shell. The behavior of these commands is identical to those of the **bsh** command, except that the following actions are not allowed:

- Changing the directory (with the **cd** command)
- Setting the value of **PATH** or **SHELL** variables
- Specifying path or command names containing a slash (/)
- Redirecting output

If the restricted shell determines that a command to be run is a shell procedure, it uses the Bourne shell to run the command. In this way, it is possible to provide an end user with shell procedures that access the full power of the Bourne shell while imposing a limited menu of commands. This situation assumes that the end user does not have write and execute permissions in the same directory.

If the *File* [*Parameter*] parameter is specified when the Bourne shell is started, the shell runs the script file identified by the *File* parameter, including any parameters specified. The script file specified must have read permission. Any **setuid** and **setgid** settings for script files are ignored. The shell then reads the commands. If using either the **-c** or **-s** flag is used, do not specify a script file.

When started with the **Rsh** command, the shell enforces restrictions after interpreting the **.profile** and **/etc/environment** files. Therefore, the writer of the **.profile** file has complete control over user actions by performing setup actions and leaving the user in an appropriate directory (probably not the login directory). An administrator can create a directory of commands in the **/usr/rbin** directory that the **Rsh** command can use by changing the PATH variable to contain the directory. If it is started with the **bsh -r** command, the shell applies restrictions when interpreting the **.profile** files.

When called with the name **Rsh**, the restricted shell reads the user's **.profile** file (**$HOME/.profile**). It acts as the regular Bourne shell while doing this, except that an interrupt causes an immediate exit instead of a return to command level.

Bourne Shell Commands

When you issue a command in the Bourne shell, it first evaluates the command and makes all indicated substitutions. It then runs the command provided that:

- The command name is a Bourne shell special built-in command.

  OR
- The command name matches the name of a defined function. If this is the case, the shell sets the positional parameters to the parameters of the function.

If the command name matches neither a built-in command nor the name of a defined function and the command names an executable file that is a compiled (binary) program, the shell (as *parent*) spawns a new (*child*) process that immediately runs the program. If the file is marked executable but is not a compiled program, the shell assumes that it is a shell procedure. In this case, the shell spawns another instance of itself (a *subshell*), to read the file and execute the commands included in it. The shell also runs a parenthesized command in a subshell. To the end user, a compiled program is run in exactly the same way as a shell procedure. The shell normally searches for commands in file system directories, in this order:

1. **/usr/bin**
2. **/etc**
3. **/usr/sbin**
4. **/usr/ucb**
5. **$HOME/bin**
6. **/usr/bin/X11**
7. **/sbin**
8. Current directory

The shell searches each directory, in turn, continuing with the next directory if it fails to find the command.

> **Note:** The **PATH** variable determines the order in which the shell searches directories. You can change the particular sequence of directories searched by resetting the **PATH** variable.

If you give a specific path name when you run a command (for example, **/usr/bin/sort**), the shell does not search any directories other than the one you specify. If the command name contains a slash (/), the shell does not use the search path.

You can give a full path name that begins with the root directory (such as **/usr/bin/sort**). You can also specify a path name relative to the current directory. If you specify, for example:

```
bin/myfile
```

the shell looks in the current directory for a directory named `bin` and in that directory for the file `myfile`.

> **Note:** The restricted shell does not run commands containing a / (slash).

The shell remembers the location in the search path of each executed command (to avoid unnecessary **exec** commands later). If it finds the command in a relative directory (one whose name does not begin with /), the shell must redetermine the command's location whenever the current directory changes. The shell forgets all remembered locations each time you change the **PATH** variable or run the **hash -r** command.

This section discusses the following:
- "Quoting Characters"
- "Signal Handling" on page 188
- "Bourne Shell Built-In Commands" on page 189
- "Command Substitution in the Bourne Shell" on page 192

Quoting Characters

Many characters have a special meaning to the shell. Sometimes you want to conceal that meaning. Single (') and double (″) quotation marks surrounding a string, or a backslash (\) before a single character allow you to conceal the character's meaning.

All characters (except the enclosing single quotation marks) are taken literally, with any special meaning removed. Thus, the command:

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

assigns the literal string `echo $? $*; ls * | wc` to the variable `stuff`. The shell does not execute the **echo**, **ls**, and **wc** commands or expand the $? and $* variables and the * (asterisk) special character.

Within double quotation marks, the special meaning of the $ (dollar sign), ` (backquote), and ″ (double quotation) characters remains in effect, while all other characters are taken literally. Thus, within double quotation marks, command and variable substitution takes place. In addition, the quotation marks do not affect the commands within a command substitution that is part of the quoted string, so characters there retain their special meanings.

Consider the following sequence:

```
ls *
file1 file2 file3
message="This directory contains `ls * ` "
echo $message
This directory contains file1 file2 file3
```

This shows that the * (asterisk) special character inside the command substitution was expanded.

To hide the special meaning of the $ (dollar sign), ` (backquote), and ″ (double quotation) characters within double quotation marks, precede these characters with a \ (backslash). When you do not use double quotation marks, preceding a character with a backslash is equivalent to placing it within single quotation marks. Hence, a backslash immediately preceding a newline character (that is, a backslash at the end of the line) hides the newline character and allows you to continue the command line on the next physical line.

Signal Handling

The shell ignores **INTERRUPT** and **QUIT** signals for an invoked command if the command is terminated with an & (ampersand); that is, if it is running in the background. Otherwise, signals have the values inherited by the shell from its parent, with the exception of the SEGMENTATION VIOLATION signal. For more information, refer to the Bourne shell built-in **trap** command.

Bourne Shell Compound Commands

A compound command is one of the following:

- Pipeline (one or more simple commands separated by the | (pipe) symbol)
- List of simple commands
- Command beginning with a reserved word
- Command beginning with the control operator ((left parenthesis).

Unless otherwise stated, the value returned by a compound command is that of the last simple command executed.

Reserved Words

The following reserved words are recognized only when they appear without quotation marks as the first word of a command:

```
for           do             done
case          esac
if            then            fi
elif          else
while         until
{             }
(             )
```

for *Identifier* [**in** *Word* . . .] **do** *List* **done**	Sets the *Identifier* parameter to the word or words specified by the *Word* parameter (one at a time) and runs the commands specified in the *List* parameter. If you omit **in** *Word* . . ., then the **for** command runs the *List* parameter for each positional parameter that is set, and processing ends when all positional parameters have been used.
case *Word* **in** *Pattern* [|*Pattern*] . . . **)** *List***;;** [*Pattern* [|*Pattern*] . . . **)** *List*;;] **. . .** **esac**	Runs the commands specified in the *List* parameter that are associated with the first *Pattern* parameter that matches the value of the *Word* parameter. Uses the same character-matching notation in patterns that are used for file name substitution, except that a / (slash), leading . (dot), or a dot immediately following a slash do not need to match explicitly.
if *List* **then** *List* [**elif** *List* **then** *List*] . . . [**else** *List*] **fi**	Runs the commands specified in the *List* parameter following the **if** command. If the command returns a zero exit value, the shell runs the *List* parameter following the first **then** command. Otherwise, it runs the *List* parameter following the **elif** command (if it exists). If this exit value is zero, the shell runs the List parameter following the next **then** command. If the command returns a non-zero exit value, the shell runs the *List* parameter following the **else** command (if it exists). If no **else** *List* or **then** *List* is performed, the **if** command returns a zero exit value.
while *List* **do** *List* **done**	Runs the commands specified in the *List* parameter following the **while** command. If the exit value of the last command in the **while** *List* is zero, the shell runs the *List* parameter following the **do** command. It continues looping through the lists until the exit value of the last command in the **while** *List* is non-zero. If no commands in the **do** *List* are performed, the **while** command returns a zero exit value.
until *List* **do** *List* **done**	Runs the commands specified in the *List* parameter following the **until** command. If the exit value of the last command in the **until** *List* is non-zero, runs the *List* following the **do** command. Continues looping through the lists until the exit value of the last command in the **until** *List* is zero. If no commands in the **do** *List* are performed, the **until** command returns a zero exit value.

(*List* **)**	Runs the commands in the *List* parameter in a subshell.
{ *List*; **}**	Runs the commands in the *List* parameter in the current shell process and does not start a subshell.
Name **()** **{** *List* **}**	Defines a function that is referenced by the *Name* parameter. The body of the function is the list of commands between the braces specified by the *List* parameter.

Bourne Shell Built-In Commands

Special commands are built in to the Bourne shell and run in the shell process. Unless otherwise indicated, output is written to file descriptor 1 (standard output) and the exit status is 0 (zero) if the command does not contain any syntax errors. Input and output redirection is permitted.

Refer to the "List of Bourne Shell Built-in Commands" on page 199 for an alphabetical listing of these commands.

The following special commands are treated somewhat differently from other special built-in commands:

```
: (colon)     exec          shift
. (dot)       exit          times
break         export        trap
continue      readonly      wait
eval          return
```

The Bourne shell processes these commands as follows:

- Keyword parameter assignment lists preceding the command remain in effect when the command completes.
- I/O redirections are processed after parameter assignments.
- Errors in a shell script cause the script to stop processing.

Special Command Descriptions

The Bourne shell provides the following special built-in commands:

Built-In Commands	
:	Returns a zero exit value.
. *File*	Reads and runs commands from the *File* parameter, and returns. Does not start a subshell. The shell uses the search path specified by the **PATH** variable to find the directory containing the specified file.
break [*n*]	Exits from the enclosing **for**, **while**, or **until** command loops, if any. If you specify the *n* variable, the **break** command breaks the number of levels specified by the *n* variable.
continue [*n*]	Resumes the next iteration of the enclosing **for**, **while**, or **until** command loops. If you specify the *n* variable, the command resumes at the *n*th enclosing loop.
cd *Directory*]	Changes the current directory to *Directory*. If you do not specify *Directory*, the value of the **HOME** shell variable is used. The **CDPATH** shell variable defines the search path for *Directory*. **CDPATH** is a colon-separated list of alternative directory names. A null path name specifies the current directory (which is the default path). This null path name appears immediately after the equal sign in the assignment or between the colon delimiters anywhere else in the path list. If *Directory* begins with a / (slash), the shell does not use the search path. Otherwise, the shell searches each directory in the **CDPATH** shell variable. **Note:** The restricted shell cannot run the **cd** shell command.
echo *String* . . .]	Writes character strings to standard output. Refer to the **echo** command for usage and parameter information. The **-n** flag is not supported.
eval [*Argument* . . .]	Reads arguments as input to the shell and runs the resulting command or commands.

Built-In Commands

exec [*Argument* . . .]	Runs the command specified by the *Argument* parameter in place of this shell without creating a new process. Input and output arguments can appear and if no other arguments appear, cause the shell input or output to be modified. This is not recommended for your login shell.
exit [*n*]	Causes a shell to exit with the exit value specified by the *n* parameter. If you omit this parameter, the exit value is that of the last command executed (the Ctrl-D key sequence also causes a shell to exit). The value of the *n* parameter can be from 0 to 255, inclusive.
export [*Name* . . .]	Marks the specified names for automatic export to the environments of subsequently executed commands. If you do not specify the *Name* parameter, the **export** command displays a list of all names that are exported in this shell. You cannot export function names.
hash [**-r**][*Command* . . .]	Finds and remembers the location in the search path of each *Command* specified. The **-r** flag causes the shell to forget all locations. If you do not specify the flag or any commands, the shell displays information about the remembered commands in the following format: `Hits   Cost   Command` `Hits` indicates the number of times a command has been run by the shell process. `Cost` is a measure of the work required to locate a command in the search path. `Command` shows the path names of each specified command. Certain situations require that the stored location of a command be recalculated; for example, the location of a relative path name when the current directory changes. Commands for which that might be done are indicated by an * (asterisk) next to the `Hits` information. `Cost` is incremented when the recalculation is done.
pwd	Displays the current directory. Refer to the **pwd** command for a discussion of command options.
read [*Name* . . .]	Reads one line from standard input. Assigns the first word in the line to the first *Name* parameter, the second word to the second *Name* parameter, and so on, with leftover words assigned to the last *Name* parameter. This command returns a value of 0 unless it encounters an end-of-file character.
readonly [*Name* . . .]	Marks the name specified by the *Name* parameter as read-only. The value of the name cannot be reset. If you do not specify any *Name*, the **readonly** command displays a list of all read-only names.
return [*n*]	Causes a function to exit with a return value of *n*. If you do not specify the *n* variable, the function returns the status of the last command performed in that function. This command is valid only when run within a shell function.

Built-In Commands

set [*Flag* [*Argument*] . . .]	Sets one or more of the following flags: **-a** Marks for export all variables to which an assignment is performed. If the assignment precedes a command name, the export attribute is effective only for that command execution environment, except when the assignment precedes one of the special built-in commands. In this case, the export attribute persists after the built-in command has completed. If the assignment does not precede a command name, or if the assignment is a result of the operation of the **getopts** or **read** commands, the export attribute persists until the variable is unset. **-e** Exits immediately if all of the following conditions exist for a command: • It exits with a return value greater than 0 (zero). • It is not part of the compound list of a **while**, **until**, or **if** command. • It is not being tested using AND or OR lists. • It is not a pipeline preceded by the ! (exclamation point) reserved word. **-f** Disables file-name substitution. **-h** Locates and remembers the commands called within functions as the functions are defined. (Normally these commands are located when the function is performed; see the **hash** command.) **-k** Places all keyword parameters in the environment for a command, not just those preceding the command name. **-n** Reads commands but does not run them. To check for shell script syntax errors, use the **-n** flag. **-t** Exits after reading and executing one command. **-u** Treats an unset variable as an error and immediately exits when performing variable substitution. An interactive shell does not exit. **-v** Displays shell input lines as they are read. **-x** Displays commands and their arguments before they are run. **—** Does not change any of the flags. This is useful in setting the **$1** positional parameter to a string beginning with a hyphen (-). Using a plus sign (+) rather than a hyphen (-) unsets flags. You can also specify these flags on the shell command line. The **$-** special variable contains the current set of flags. Any *Argument* to the **set** command becomes a positional parameter and is assigned, in order, to **$1**, **$2**, and so on. If you do not specify a *flag* or *Argument*, the **set** command displays all the names and values of the current shell variables.
shift [*n*]	Shifts command line arguments to the left; that is, reassigns the value of the positional parameters by discarding the current value of **$1** and assigning the value of **$2** to **$1**, of **$3** to **$2**, and so on. If there are more than 9 command line arguments, the 10th is assigned to **$9** and any that remain are still unassigned (until after another **shift**). If there are 9 or fewer arguments, the **shift** command unsets the highest-numbered positional parameter that has a value. The **$0** positional parameter is never shifted. The **shift** *n* command is a shorthand notation specifying *n* number of consecutive shifts. The default value of the *n* parameter is 1.
test *Expression* | [*Expression*]	Evaluates conditional expressions. Refer to the **test** command for a discussion of command flags and parameters. The **-h** flag is not supported by the built-in test command in **bsh**.
times	Displays the accumulated user and system times for processes run from the shell.

Built-In Commands

trap [*Command*] [*n*] . . .
: Runs the command specified by the *Command* parameter when the shell receives the signal or signals specified by the *n* parameter. The **trap** commands are run in order of signal number. Any attempt to set a trap on a signal that was ignored on entry to the current shell is ineffective.

> **Note:** The shell scans the *Command* parameter once when the trap is set and again when the trap is taken.

If you do not specify a command, then all traps specified by the *n* parameter are reset to their current values. If you specify a null string, this signal is ignored by the shell and by the commands it invokes. If the *n* parameter is zero (0), the specified command is run when you exit from the shell. If you do not specify either a command or a signal, the **trap** command displays a list of commands associated with each signal number.

type [*Name* . . .]
: For each *Name* specified, indicates how the shell would interpret it as a command name.

ulimit [**-HS**] [**-c** | **-d** | **-f** | **-m** | **-s** | **-t**] [*limit*]
: Displays or adjusts allocated shell resources. The shell resource settings can be displayed either individually or as a group. The default mode is to display resources set to the soft setting, or the lower bound, as a group.

The setting of shell resources depends on the effective user ID of the current shell. The hard level of a resource can be set only if the effective user ID of the current shell is root. You will get an error if you are not root user and you are attempting to set the hard level of a resource. By default, the root user sets both the hard and soft limits of a particular resource. The root user should therefore be careful in using the **-S**, **-H**, or default flag usage of limit settings. Unless you are a root user, you can set only the soft limit of a resource. After a limit has been decreased by a non-root user, it cannot be increased, even back to the original system limit.

To set a resource limit, select the appropriate flag and the limit value of the new resource, which should be an integer. You can set only one resource limit at a time. If more than one resource flag is specified, you receive undefined results. By default, **ulimit** with only a new value on the command line sets the file size of the shell. Use of the **-f** flag is optional.

You can specify the following **ulimit** command flags:

-c	Sets or displays core segment for shell.
-d	Sets or displays data segment for shell.
-f	Sets or displays file size for shell.
-H	Sets or displays hard resource limit (root user only)
-m	Sets or displays memory for shell.
-s	Sets or displays stack segment for shell.
-S	Sets or displays soft resource limit.
-t	Sets or displays CPU time maximum for shell.

umask [*nnn*]
: Determines file permissions. This value, along with the permissions of the creating process, determines a file's permissions when the file is created. The default is 022. When no value is entered, umask displays the current value.

unset [*Name* . . .]
: Removes the corresponding variable or function for each name specified by the *Name* parameter. The **PATH**, **PS1**, **PS2**, **MAILCHECK**, and **IFS** shell variables cannot be unset.

wait [*n*]
: Waits for the child process whose process number is specified by the *n* parameter to exit and then returns the exit status of that process. If you do not specify the *n* parameter, the shell waits for all currently active child processes and the return value is 0.

Command Substitution in the Bourne Shell

Command substitution allows you to capture the output of any command as an argument to another command. When you place a command line within backquotes (``), the shell first runs the command or commands, and then replaces the entire expression, including the backquotes, with the output. This feature is often used to give values to shell variables. For example, the statement:

```
today=`date`
```

assigns the string representing the current date to the `today` variable. The following assignment saves, in the `files` variable, the number of files in the current directory:

```
files=`ls | wc -l`
```

You can perform command substitution on any command that writes to standard output.

To nest command substitutions, precede each of the nested backquotes with a backslash (\), as in:

```
logmsg=`echo Your login directory is \`pwd\``
```

You can also give values to shell variables indirectly by using the **read** special command. This command takes a line from standard input (usually your keyboard) and assigns consecutive words on that line to any variables named. For example:

```
read first init last
```

takes an input line of the form:

```
J. Q. Public
```

and has the same effect as if you had typed:

```
first=J. init=Q. last=Public
```

The **read** special command assigns any excess words to the last variable.

Variable and File-Name Substitution in the Bourne Shell

The Bourne shell permits you to do variable and file-name substitutions.

The following sections discuss creating and substituting variables in the Bourne shell:

- "Variable Substitution in the Bourne Shell"
- "User-Defined Variables"
- "Conditional Substitution" on page 196
- "Positional Parameters" on page 197
- "File-Name Substitution in the Bourne Shell" on page 198
- "Character Classes" on page 198

Variable Substitution in the Bourne Shell

The Bourne shell has several mechanisms for creating variables (assigning a string value to a name). Certain variables, positional parameters and keyword parameters are normally set only on a command line. Other variables are simply names to which you or the shell can assign string values.

User-Defined Variables

The shell recognizes alphanumeric variables to which string values can be assigned. To assign a string value to a name, type:

```
Name=String
```

Press Enter.

A name is a sequence of letters, digits, and underscores that begins with an underscore or a letter. To use the value that you have assigned to a variable, add a dollar sign ($) to the beginning of its name. Thus, the *$Name* variable yields the value specified by the *String* variable. Note that no spaces are on either side of the equal sign (=) in an assignment statement. (Positional parameters cannot appear in an assignment statement. They can be set only as described in “Positional Parameters” on page 197.) You can put more than one assignment on a command line, but remember that the shell performs the assignments from right to left.

If you enclose the *String* variable with double or single quotation marks (″ or ’), the shell does not treat blanks, tabs, semicolons, and newline characters within the string as word delimiters, but imbeds them literally in the string.

If you enclose the *String* variable with double quotation marks (″), the shell still recognizes variable names in the string and performs variable substitution; that is, it replaces references to positional parameters and other variable names that are prefaced by dollar sign ($) with their corresponding values, if any. The shell also performs command substitution within strings that are enclosed in double quotation marks.

If you enclose the *String* variable with single quotation marks (’), the shell does not substitute variables or commands within the string. The following sequence illustrates this difference:

```
You:           num=875
               number1="Add $num"
               number2='Add $num'
               echo $number1
System:        Add 875
You:           echo $number2
System:        Add $num
```

The shell does not reinterpret blanks in assignments after variable substitution. Thus, the following assignments result in `$first` and `$second` having the same value:

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

When you reference a variable, you can enclose the variable name (or the digit designating a positional parameter) in { } to delimit the variable name from any string following. In particular, if the character immediately following the name is a letter, digit, or underscore, and the variable is not a positional parameter, then the braces are required:

```
You:           a='This is a'
               echo "${a}n example"
System:        This is an example
You:           echo "$a test"
System:        This is a test
```

Refer to “Conditional Substitution” on page 196 for a different use of braces in variable substitutions.

Variables Used by the Shell

The shell uses the following variables. Although the shell sets some of them, you can set or reset all of them:

CDPATH	Specifies the search path for the **cd** (change directory) command.
HOME	Indicates the name of your *login directory*, the directory that becomes the current directory upon completion of a login. The **login** program initializes this variable. The **cd** command uses the value of the **$HOME** variable as its default value. Using this variable rather than an explicit path name in a shell procedure allows the procedure to be run from a different directory without alterations.
IFS	The characters that are IFS (internal field separators), the characters that the shell uses during blank interpretation; see “Blank Interpretation” on page 196. The shell initially sets the **IFS** variable to include the blank, tab, and newline characters.

LANG	Determines the locale to use for the locale categories when both the **LC_ALL** variable and the corresponding environment variable (beginning with **LC_**) do not specify a locale. For more information about locales, see "Locale Overview" in *AIX 5L Version 5.2 National Language Support Guide and Reference*.
LC_ALL	Determines the locale to be used to override any values for locale categories specified by the settings of the **LANG** environment variable or any environment variables beginning with **LC_**.
LC_COLLATE	Defines the collating sequence to use when sorting names and when character ranges occur in patterns.
LC_CTYPE	Determines the locale for the interpretation of sequences of bytes of text data as characters (that is, single- versus multibyte characters in arguments and input files), which characters are defined as letters (**alpha** character class), and the behavior of character classes within pattern matching.
LC_MESSAGES	Determines the language in which messages should be written.
LIBPATH	Specifies the search path for shared libraries.
LOGNAME	Specifies your login name, marked **readonly** in the **/etc/profile** file.
MAIL	Indicates the path name of the file used by the mail system to detect the arrival of new mail. If this variable is set, the shell periodically checks the modification time of this file and displays the value of **$MAILMSG** if the time changes and the length of the file is greater than 0. Set the **MAIL** variable in the **.profile** file. The value normally assigned to it by users of the **mail** command is **/usr/spool/mail/$LOGNAME**.
MAILCHECK	The number of seconds that the shell lets elapse before checking again for the arrival of mail in the files specified by the **MAILPATH** or **MAIL** variables. The default value is 600 seconds (10 minutes). If you set the **MAILCHECK** variable to 0, the shell checks before each prompt.
MAILMSG	The mail notification message. If you explicitly set the **MAILMSG** variable to a null string (`MAILMSG=""`), no message is displayed.
MAILPATH	A list of file names separated by colons. If this variable is set, the shell informs you of the arrival of mail in any of the files specified in the list. You can follow each file name by a `%` and a message to be displayed when mail arrives. Otherwise, the shell uses the value of the **MAILMSG** variable or, by default, the message `[YOU HAVE NEW MAIL]`. **Note:** When the **MAILPATH** variable is set, these files are checked instead of the file set by the **MAIL** variable. To check the files set by the **MAILPATH** variable and the file set by the **MAIL** variable, specify the **MAIL** file in your list of **MAILPATH** files.
PATH	The search path for commands, which is an ordered list of directory path names separated by colons. The shell searches these directories in the specified order when it looks for commands. A null string anywhere in the list represents the current directory. The **PATH** variable is normally initialized in the **/etc/environment** file, usually to **/usr/bin:/etc:/usr/sbin:/usr/ucb:/usr/bin/X11:/sbin**. You can reset this variable to suit your own needs. The **PATH** variable provided in your **.profile** file also includes **$HOME/bin** and your current directory. If you have a project-specific directory of commands, for example, **/project/bin**, that you want searched before the standard system directories, set your **PATH** variable as follows: `PATH=/project/bin:$PATH` The best place to set your **PATH** variable to a value other than the default value is in your **$HOME/.profile** file. You cannot reset the **PATH** variable if you are executing commands under the restricted shell.
PS1	The string to be used as the primary system prompt. An interactive shell displays this prompt string when it expects input. The default value of the **PS1** variable is $ followed by a blank space, for nonroot users.
PS2	The value of the secondary prompt string. If the shell expects more input when it encounters a new-line character in its input, it prompts with the value of the **PS2** variable. The default value of the **PS2** variable is > , followed by a blank space.
SHACCT	The name of a file that you own. If this variable is set, the shell writes an accounting record in the file for each shell script executed. You can use accounting programs such as **acctcom** and **acctcms** to analyze the data collected.

SHELL	The path name of the shell, which is kept in the environment. This variable should be set and exported by the **$HOME/.profile** file of each restricted login.
TIMEOUT	The number of minutes a shell remains inactive before it exits. If this variable is set to a value greater than zero (0), the shell exits if a command is not entered within the prescribed number of seconds after issuing the **PS1** prompt. (Note that the shell can be compiled with a maximum boundary that cannot be exceeded for this value.) A value of zero indicates no time limit.

Predefined Special Variables

Several variables have special meanings. The following variables are set only by the shell.

$@ Expands the positional parameters, beginning with **$1**. Each parameter is separated by a space.

If you place ″ ″ around **$@**, the shell considers each positional parameter a separate string. If no positional parameters exist, the Bourne shell expands the statement to an unquoted null string.

$* Expands the positional parameters, beginning with **$1**. The shell separates each parameter with the first character of the **IFS** variable value.

If you place ″ ″ around **$***, the shell includes the positional parameter values, in double quotation marks. Each value is separated by the first character of the **IFS** variable.

$#** Specifies the number of positional parameters passed to the shell, not counting the name of the shell procedure itself. The **$# variable thus yields the number of the highest-numbered positional parameter that is set. One of the primary uses of this variable is to check for the presence of the required number of arguments. Only positional parameters **$0** through **$9** are accessible through the shell. See ″"Positional Parameters" on page 197 for more information.

$?** Specifies the exit value of the last command executed. Its value is a decimal string. Most commands return a value of 0 to indicate successful completion. The shell itself returns the current value of the **$? variable as its exit value.

$$ Identifies the process number of the current process. Because process numbers are unique among all existing processes, this string is often used to generate unique names for temporary files.

The following example illustrates the recommended practice of creating temporary files in a directory used only for that purpose:

```
temp=/tmp/$$
ls >$temp
.
.
.
rm $temp
```

$! Specifies the process number of the last process run in the background using the **&** terminator.

$- A string consisting of the names of the execution flags currently set in the shell.

Blank Interpretation

After the shell performs variable and command substitution, it scans the results for internal field separators (those defined in the **IFS** shell variable). The shell splits the line into distinct words at each place it finds one or more of these characters separating each distinct word with a single space. It then retains explicit null arguments (″″ or '') and discards implicit null arguments (those resulting from parameters that have no values).

Conditional Substitution

Normally, the shell replaces the expression $*Variable* with the string value assigned to the *Variable* variable, if there is one. However, there is a special notation that allows *conditional substitution*, depending on whether the variable is set or not null, or both. By definition, a variable is set if it has ever been assigned a value. The value of a variable can be the null string, which you can assign to a variable in any one of the following ways:

```
A=
```

`bcd=""`

`Efg=''`	Assigns the null string to the `A`, `bcd`, and `Efg`.
`set '' ""`	Sets the first and second positional parameters to the null string and unsets all other positional parameters.

The following is a list of the available expressions you can use to perform conditional substitution:

${*Variable***-** *String***}**	If the variable is set, substitute the *Variable* value in place of this expression. Otherwise, replace this expression with the *String* value.
${*Variable***:-***String***}**	If the variable is set and not null, substitute the *Variable* value in place of this expression. Otherwise, replace this expression with the *String* value.
${*Variable***=***String***}**	If the variable is set, substitute the *Variable* value in place of this expression. Otherwise, set the *Variable* value to the *String* value and then substitute the *Variable* value in place of this expression. You cannot assign values to positional parameters in this fashion.
${*Variable***:=***String***}**	If the variable is set and not null, substitute the *Variable* value in place of this expression. Otherwise, set the *Variable* value to the *String* value and then substitute the *Variable* value in place of this expression. You cannot assign values to positional parameters in this fashion.
${*Variable***?***String***}**	If the variable is set, substitute the *Variable* value in place of this expression. Otherwise, display a message of the following form: `Variable: String` and exit from the current shell (unless the shell is the login shell). If you do not specify a value for the *String* variable, the shell displays the following message: `Variable: parameter null or not set`
${*Variable***:?***String***}**	If the variable is set and not null, substitute the *Variable* value in place of this expression. Otherwise, display a message of the following form: `Variable: String` and exit from the current shell (unless the shell is the login shell). If you do not specify the *String* value, the shell displays the following message: `Variable: parameter null or not set`
${*Variable***+***String***}**	If the variable is set, substitute the *String* value in place of this expression. Otherwise, substitute the null string.
${*Variable***:+***String***}**	If the variable is set and not null, substitute the *String* value in place of this expression. Otherwise, substitute the null string.

In conditional substitution, the shell does not evaluate the *String* variable until the shell uses this variable as a substituted string. Thus, in the following example, the shell executes the **pwd** command only if `d` is not set or is null:

```
echo ${d:-`pwd`}
```

Positional Parameters

When you run a shell procedure, the shell implicitly creates positional parameters that reference each word on the command line by its position on the command line. The word in position 0 (the procedure name) is called **$0**, the next word (the first parameter) is called **$1**, and so on, up to **$9**. To refer to command line parameters numbered higher than 9, use the built-in **shift** command.

You can reset the values of the positional parameters explicitly by using the built-in **set** command.

Note: When an argument for a position is not specified, its positional parameter is set to null. Positional parameters are global and can be passed to nested shell procedures.

File-Name Substitution in the Bourne Shell

Command parameters are often file names. You can automatically produce a list of file names as parameters on a command line. To do this, specify a character that the shell recognizes as a pattern-matching character. When a command includes such a character, the shell replaces it with the file names in a directory.

> **Note:** The Bourne shell does not support file-name expansion based on equivalence classification of characters.

Most characters in such a pattern match themselves, but you can also use some special pattern-matching characters in your pattern. These special characters are as follows:

*****	Matches any string, including the null string
?	Matches any one character
[. . .]	Matches any one of the characters enclosed in square brackets
[! . . .]	Matches any character within square brackets *other than* one of the characters that follow the exclamation mark

Within square brackets, a pair of characters separated by a - specifies the set of all characters lexicographically within the inclusive range of that pair, according to the binary ordering of character values.

Pattern matching has some restrictions. If the first character of a file name is a dot (.), it can be matched only by a pattern that also begins with a dot. For example, * matches the file names **myfile** and **yourfile** but not the file names **.myfile** and **.yourfile**. To match these files, use a pattern such as the following:

```
.*file
```

If a pattern does not match any file names, then the pattern itself is returned as the result of the attempted match.

File and directory names should not contain the characters *, ?, [, or] because they can cause infinite recursion (that is, infinite loops) during pattern-matching attempts.

Character Classes

You can also use character classes to match file names, as follows:

```
[[:charclass:]]
```

This format instructs the system to match any single character belonging to the specified class. The defined classes correspond to **ctype** subroutines, as follows:

Character Class	**Definition**
alnum	Alphanumeric characters
alpha	Uppercase and lowercase letters
blank	Space or horizontal tab
cntrl	Control characters
digit	Digits
graph	Graphic characters
lower	Lowercase letters
print	Printable characters
punct	Punctuation characters
space	Space, horizontal tab, carriage return, newline, vertical tab or form-feed character
upper	Uppercase characters
xdigit	Hexadecimal digits

Input and Output Redirection in the Bourne Shell

In general, most commands do not know whether their input or output is associated with the keyboard, the display screen, or a file. Thus, a command can be used conveniently either at the keyboard or in a pipeline.

The following redirection options can appear anywhere in a simple command. They can also precede or follow a command, but are not passed to the command.

<*File*	Uses the specified file as standard input.
>*File*	Uses the specified file as standard output. Creates the file if it does not exist; otherwise, truncates it to zero length.
> >*File*	Uses the specified file as standard output. Creates the file if it does not exist; otherwise, adds the output to the end of the file.
<<[**-**]*eofstr*	Reads as standard input all lines from the *eofstr* variable up to a line containing only *eofstr* or up to an end-of-file character. If any character in the *eofstr* variable is quoted, the shell does not expand or interpret any characters in the input lines. Otherwise, it performs variable and command substitution and ignores a quoted newline character (**\newline**). Use a \ to quote characters within the *eofstr* variable or within the input lines. If you add a - to the << redirection option, then all leading tabs are stripped from the *eofstr* variable and from the input lines.
<&*Digit*	Associates standard input with the file descriptor specified by the *Digit* variable.
>&*Digit*	Associates standard output with the file descriptor specified by the *Digit* variable.
<&-	Closes standard input.
>&-	Closes standard output.

Note: The restricted shell does not allow output redirection.

For more information about redirection, see Chapter 5, "Input and Output Redirection" on page 45.

List of Bourne Shell Built-in Commands

:	Returns a zero exit value
.	Reads and executes commands from a file parameter and then returns.
break	Exists from the enclosing **for**, **while**, or **until** command loops, if any.
cd	Changes the current directory to the specified directory.
continue	Resumes the next iteration of the enclosing **for**, **while**, or **until** command loops.
echo	Writes character strings to standard output.
eval	Reads the arguments as input to the shell and executes the resulting command or commands.
exec	Executes the command specified by the *Argument* parameter, instead of this shell, without creating a new process.
exit	Exits the shell whose exit status is specified by the *n* parameter.
export	Marks names for automatic export to the environment of subsequently executed commands.
hash	Finds and remembers the location in the search path of specified commands.
pwd	Displays the current directory.
read	Reads one line from standard input.
readonly	Marks name specified by *Name* parameter as read-only.
return	Causes a function to exit with a specified return value.
set	Controls the display of various parameters to standard output.
shift	Shifts command-line arguments to the left.
test	Evaluates conditional expressions.
times	Displays the accumulated user and system times for processes run from the shell.
trap	Runs a specified command when the shell receives a specified signal or signals.
type	Interprets how the shell would interpret a specified name as a command name.

ulimit	Displays or adjusts allocated shell resources.
umask	Determines file permissions.
unset	Removes the variable or function corresponding to a specified name.
wait	Waits for the specified child process to end and reports its termination status.

C Shell

The C shell is an interactive command interpreter and a command programming language. It uses syntax that is similar to the C programming language. The **csh** command starts the C shell.

When you log in, the **csh** command first searches the systemwide setup file **/etc/csh.cshrc.** If the setup file is there, the C shell executes the commands stored in that file. Next, the C shell executes the systemwide setup file **/etc/csh.login** if it is available. Then, it searches your home directory for the **.cshrc** and **.login** files. If they exist, they contain any customized user information pertinent to running the C shell. All variables set in the **/etc/csh.cshrc** and **/etc/csh.login** files might be overridden by your **.cshrc** and **.login** files in your **$HOME** directory. Only the root user can modify the **/etc/csh.cshrc** and **/etc/csh.login** files.

The **/etc/csh.login** and **$HOME/.login** files are executed only once at login time. These files are generally used to hold environment variable definitions, commands that you want executed once at login, or commands that set up terminal characteristics.

The **/etc/csh.cshrc** and **$HOME/.cshrc** files are executed at login time, and every time the **csh** command or a C shell script is invoked. They are generally used to define C shell characteristics like aliases and C shell variables (for example, history, noclobber, or ignoreeof). It is recommended that you only use the C Shell built-in commands (see “C Shell Built-In Commands” on page 202) in the **/etc/csh.cshrc** and **$HOME/.cshrc** files because using other commands increases the startup time for shell scripts.

This section discusses the following:
- “C Shell Limitations” on page 201
- “Signal Handling” on page 201
- “C Shell Commands” on page 201
  - “C Shell Built-In Commands” on page 202
  - “C Shell Expressions and Operators” on page 207
  - “Command Substitution in the C Shell” on page 208
  - “Nonbuilt-in C Shell Command Execution” on page 209
- “History Substitution in the C Shell” on page 209
  - “History Lists” on page 209
  - “Event Specification” on page 210
  - “Quoting with Single and Double Quotes” on page 211
- “Alias Substitution in the C Shell” on page 212
- “Variable and File-Name Substitution in the C Shell” on page 213
  - “Variable Substitution in the C Shell” on page 213
  - “File-Name Substitution in the C Shell” on page 214
  - “File-Name Expansion” on page 214
  - “File-Name Abbreviation” on page 215
  - “Character Classes” on page 216
- “Environment Variables in the C Shell” on page 216
- “Input and Output Redirection in the C Shell” on page 218

- "Job Control in the C Shell" on page 219
- "C Shell" on page 221

C Shell Limitations

The following are limitations of the C shell:

- Words can be no longer than 1024 bytes.
- Argument lists are limited to ARG_MAX bytes. Values for the ARG_MAX variable are found in the **/usr/include/sys/limits.h** file.
- The number of arguments to a command that involves file-name expansion is limited to 1/6th the number of bytes allowed in an argument list.
- Command substitutions can substitute no more bytes than are allowed in an argument list.
- To detect looping, the shell restricts the number of alias substitutions on a single line to 20.
- The **csh** command does not support file-name expansion based on equivalence classification of characters.
- File descriptors (other than standard in, standard out, and standard error) opened before **csh** executes any application are not available to that application.

Signal Handling

The C shell normally ignores quit signals. Jobs running detached are not affected by signals generated from the keyboard (**INTERRUPT**, **QUIT**, and **HANGUP**). Other signals have the values the shell inherits from its parent. You can control the shell's handling of **INTERRUPT** and **TERMINATE** signals in shell procedures with **onintr**. Login shells catch or ignore **TERMINATE** signals depending on how they are set up. Shells other than login shells pass **TERMINATE** signals on to the child processes. In no cases are **INTERRUPT** signals allowed when a login shell is reading the **.logout** file.

C Shell Commands

A simple command is a sequence of words separated by blanks or tabs.

A *word* is a sequence of characters or numerals, or both, that does not contain blanks without quotation marks. In addition, the following characters and doubled characters also form single words when used as command separators or terminators:

```
&       |       ;
&&      ||      <<       > >
<       >       (         )
```

These special characters can be parts of other words. Preceding them with a \, however, prevents the shell from interpreting them as special characters. Strings enclosed in ' ' or " " (matched pairs of quotation characters) or backquotes can also form parts of words. Blanks, tab characters, and special characters do not form separate words when they are enclosed in these marks. In addition, you can enclose a newline character within these marks by preceding it with a \.

The first word in the simple command sequence (numbered 0) usually specifies the name of a command. Any remaining words, with a few exceptions, are passed to that command. If the command specifies an executable file that is a compiled program, the shell immediately runs that program. If the file is marked executable but is not a compiled program, the shell assumes that it is a shell script. In this case, the shell starts another instance of itself (a subshell) to read the file and execute the commands included in it.

This section discusses the following:

- "C Shell Built-In Commands" on page 202

- "C Shell Expressions and Operators" on page 207
- "Command Substitution in the C Shell" on page 208
- "Nonbuilt-in C Shell Command Execution" on page 209

C Shell Built-In Commands

Built-in commands are run within the shell. If a built-in command occurs as any component of a pipeline, except the last, the command runs in a subshell.

> **Note:** If you enter a command from the C shell prompt, the system searches for a built-in command first. If a built-in command does not exist, the system searches the directories specified by the **path** shell variable for a system-level command. Some C shell built-in commands and operating system commands have the same name. However, these commands do not necessarily work the same way. For more information on how the command works, check the appropriate command description.

If you run a shell script from the shell and the first line of the shell script begins with `#!/ShellPathname`, the C shell runs the shell specified in the comment to process the script. Otherwise, it runs the default shell (the shell linked to **/usr/bin/sh**). If run by the default shell, C shell built-in commands might not be recognized. To run C shell commands, make the first line of the script `#!/usr/bin/csh`.

Refer to the "List of C Shell Built-in Commands" on page 219 for an alphabetic listing of the built-in commands.

C Shell Command Descriptions

The C shell provides the following built-in commands:

alias [*Name* [*WordList*]]	Displays all aliases if you do not specify any parameters. Otherwise, the command displays the alias for the specified *Name*. If *WordList* is specified, this command assigns the value of *WordList* to the alias *Name*. The specified alias *Name* cannot be **alias** or **unalias**.
bg [**%***Job* ...]	Puts the current job or job specified by *Job* into the background, continuing the job if it was stopped.
break	Resumes running after the **end** of the nearest enclosing **foreach** or **while** command.
breaksw	Breaks from a **switch** command; resumes after the **endsw** command.
case *Label***:**	Defines a *Label* in a **switch** command.
cd[*Name*]	Equivalent to the **chdir** command (see following description).
chdir [*Name*]	Changes the current directory to that specified by the *Name* variable. If you do not specify *Name*, the command changes to your home directory. If the value of the *Name* variable is not a subdirectory of the current directory and does not begin with /, ./, or ../, the shell checks each component of the **cdpath** shell variable to see if it has a subdirectory matching the *Name* variable. If the *Name* variable is a shell variable with a value that begins with /, the shell tries this to see if it is a directory. The **chdir** command is equivalent to the **cd** command.
continue	Continues execution at the **end** of the nearest enclosing **while** or **foreach** command.
default:	Labels the **default** case in a **switch** statement. The **default** should come after all other **case** labels.
dirs	Displays the directory stack.
echo	Writes character strings to the standard output of the shell.
else	Runs the commands that follow the second **else** in an **if (***Expression***) then** ...**else if (***Expression2***) then** ... **else** ... **endif** command sequence.

end	Successively sets the *Name* variable to each member specified by the *List* variable and runs the sequence of *Commands* between the **foreach** and the matching **end** statements. The **foreach** and **end** statements must appear alone on separate lines. Uses the **continue** statement to continue the loop and the **break** statement to end the loop prematurely. When the **foreach** command is read from the terminal, the C shell prompts with a ? to allow *Commands* to be entered. Commands within loops, prompted for by ?, are not placed in the history list.
endif	If the *Expression* variable is true, runs the *Commands* that follow the first **then** statement. If the **else if** *Expression2* is true, runs the *Commands* that follow the second **then** statement. If the **else if** *Expression2* is false, runs the *Commands* that follow the **else**. Any number of **else if** pairs are possible. Only one **endif** statement is needed. The **else** segment is optional. The words **else** and **endif** can be used only at the beginning of input lines. The **if** segment must appear alone on its input line or after an **else** command.
endsw	Successively matches each **case** label against the value of the *string* variable. The *string* is command and file name expanded first. Use the pattern-matching characters *, ?, and [. . .] in the **case** labels, which are variable-expanded. If none of the labels match before a **default** label is found, the execution begins after the **default** label. The **case** label and the **default** label must appear at the beginning of the line. The **breaksw** command causes execution to continue after the **endsw** command. Otherwise, control might fall through the **case** and **default** labels, as in the C programming language. If no label matches and there is no **default**, execution continues after the **endsw** command.
eval *Parameter* . . .	Reads the value of the *Parameter* variable as input to the shell and runs the resulting command or commands in the context of the current shell. Use this command to run commands generated as the result of command or variable substitution, since parsing occurs before these substitutions.
exec *Command*	Runs the specified *Command* in place of the current shell.
exit [**(***Expression***)**	Exits the shell with either the value of the **status** shell variable (if no *Expression* is specified) or with the value of the specified *Expression*.
fg [%*Job* ...]	Brings the current job or job specified by *Job* into the foreground, continuing the job if it was stopped.
foreach *Name* (*List*) *Command*. . .	Successively sets a *Name* variable for each member specified by the *List* variable and a sequence of commands, until reaching an **end** command.
glob *List*	Displays *List* using history, variable, and file name expansion. Puts a null character between words and does not include a carriage return at the end.
goto *Word*	Continues to run after the line specified by the *Word* variable. The specified *Word* is file name and command expanded to yield a string of the form specified by the *Label:* variable. The shell rewinds its input as much as possible and searches for a line of the form *Label***:**, possibly preceded by blanks or tabs.
hashstat	Displays statistics indicating how successful the hash table has been at locating commands.
history [**-r** | **-h**] [*n*]	Displays the history event list. The oldest events are displayed first. If you specify a number *n*, only the specified number of the most recent events are displayed. The **-r** flag reverses the order in which the events are displayed so the most recent is displayed first. The **-h** flag displays the history list without leading numbers. Use this flag to produce files suitable for use with the **-h** flag of the **source** command.

if (*Expression*) *Command*	Runs the specified *Command* (including its arguments) if the specified *Expression* is true. Variable substitution on the *Command* variable happens early, at the same time as the rest of the **if** statement. The specified *Command* must be a simple command (rather than a pipeline, command list, or parenthesized command list). **Note:** Input and output redirection occurs even if the *Expression* variable is false and the *Command* is not executed.
jobs [**-l**]	Lists the active jobs. With the **-l** (lowercase *L*) flag, the **jobs** command lists process IDs in addition to the job number and name.
kill -l | [[-*Signal*] % *Job*...|*PID*...]	Sends either the **TERM** (terminate) signal or the signal specified by *Signal* to the specified *Job* or *PID* (process). Specify signals either by number or by name (as given in the **/usr/include/sys/signal.h** file, stripped of the **SIG** prefix). The **-l** (lowercase *L*) flag lists the signal names.
limit [**-h**] [*Resource* [*Max-Use*]]	Limits the usage of the specified resource by the current process and each process it creates. Process resource limits are defined in the **/etc/security/limits** file. Controllable resources are the central processing unit (CPU) time, file size, data size, core dump size, and memory use. Maximum allowable values for these resources are set with the **mkuser** command when the user is added to the system. They are changed with the **chuser** command. Limits are categorized as either soft or hard. Users may increase their soft limits up to the ceiling imposed by the hard limits. You must have root user authority to increase a soft limit above the hard limit, or to change hard limits. The **-h** flag displays hard limits instead of the soft limits. If a *Max-Use* parameter is not specified, the **limit** command displays the current limit of the specified resource. If the *Resource* parameter is not specified, the **limit** command displays the current limits of all resources. For more information about the resources controlled by the **limit** subcommand, see the **getrlimit**, **setrlimit**, or **vlimit** subroutine in the *AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1*. The *Max-Use* parameter for CPU time is specified in the hh:mm:ss format. The *Max-Use* parameter for other resources is specified as a floating-point number or an integer optionally followed by a scale factor. The scale factor is: k or kilobytes (1024 bytes), m or megabytes, or b or blocks (the units used by the **ulimit** subroutine as explained in the *AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2*). If you do not specify a scale factor, k is assumed for all resources. For both resource names and scale factors, unambiguous prefixes of the names suffice. **Note:** This command limits the physical memory (memory use) available for a process only if there is contention for system memory by other active processes.
login	Ends a login shell and replaces it with an instance of the **/usr/bin/login** command. This is one way to log out (included for compatibility with the **ksh** and **bsh** commands).
logout	Ends a login shell. This command must be used if the **ignoreeof** option is set.
nice [**+***n*] [*Command*]	If no values are specified, sets the priority of commands run in this shell to 24. If the **+***n* flag is specified, sets the priority plus the specified number. If the +n flag and *Command* are specified, runs *Command* at priority 24 plus the specified number. If you have root user authority, you can run the **nice** statement with a negative number. The *Command* always runs in a subshell, and the restrictions placed on commands in simple **if** statements apply.

nohup [*Command*]	Causes **hangups** to be ignored for the remainder of the script when no *Command* is specified. If *Command* is specified, causes the specified *Command* to be run with **hangups** ignored. To run a pipeline or list of commands, put the pipeline or list in a shell script, give the script execute permission, and use the shell script as the value of the *Command* variable. All processes run in the background with & are effectively protected from being sent a **hangup** signal when you log out. However, these processes are still subject to explicitly sent **hangups** unless the **nohup** statement is used.
notify [**%***Job*...]	Causes the shell to notify you asynchronously when the status of the current job or specified *Job* changes. Normally, the shell provides notification just before it presents the shell prompt. This feature is automatic if the **notify** shell variable is set.
onintr [**-** | *Label*]	Controls the action of the shell on interrupts. If no arguments are specified, restores the default action of the shell on interrupts, which ends shell scripts or returns to the command input level. If a **-** flag is specified, causes all interrupts to be ignored. If *Label* is specified, causes the shell to run a **goto** *Label* statement when the shell receives an interrupt or when a child process ends due to an interruption. In any case, if the shell is running detached and interrupts are being ignored, all forms of the **onintr** statement have no meaning. Interrupts continue to be ignored by the shell and all invoked commands.
popd [**+***n*]	Pops the directory stack and changes to the new top directory. If you specify a +*n* variable, the command discards the *n*th entry in the stack. The elements of the directory stack are numbered from the top, starting at 0.
pushd [**+***n*|*Name*]	With no arguments, exchanges the top two elements of the directory stack. With the *Name* variable, the command changes to the new directory and pushes the old current directory (as given in the **cwd** shell variable) onto the directory stack. If you specify a +*n* variable, the command rotates the *n*th component of the directory stack around to be the top element and changes to it. The members of the directory stack are numbered from the top, starting at 0.
rehash	Causes recomputation of the internal hash table of the contents of the directories in the **path** shell variable. This action is needed if new commands are added to directories in the **path** shell variable while you are logged in. The **rehash** command is necessary only if commands are added to one of the user's own directories or if someone changes the contents of one of the system directories.
repeat *Count Command*	Runs the specified *Command*, subject to the same restrictions as commands in simple **if** statements, the number of times specified by *Count*. **Note:** I/O redirections occur exactly once, even if the *Count* variable equals 0.
set [[*Name*[n]] [**=** *Word*]] | [*Name* = (*List*)]	Shows the value of all shell variables when used with no arguments. Variables that have more than a single word as their value are displayed as a parenthesized word list. If only *Name* is specified, the C shell sets the *Name* variable to the null string. Otherwise, sets *Name* to the value of the *Word* variable, or sets the *Name* variable to the list of words specified by the *List* variable. When *n* is specified, the *n*th component of the *Name* variable is set to the value of the *Word* variable; the *n*th component must already exist. In all cases, the value is command and file name expanded. These arguments may be repeated to set multiple values in a single **set** command. However, variable expansion happens for all arguments before any setting occurs.
setenv*Name Value*	Sets the value of the environment variable specified by the *Name* variable to *Value*, a single string. The most commonly used environment variables, **USER**, **TERM**, **HOME**, and **PATH**, are automatically imported to and exported from the C shell variables **user**, **term**, **home**, and **path**. There is no need to use the **setenv** statement for these.

shift [*Variable*]
Shifts the members of the **argv** shell variable or the specified *Variable* to the left. An error occurs if the **argv** shell variable or specified *Variable* is not set or has less than one word as its value.

source[**-h**] *Name*
Reads commands specified by the *Name* variable. You can nest the **source** commands. However, if they are nested too deeply, the shell might run out of file descriptors. An error in a **source** command at any level ends all nested **source** commands. Normally, input during **source** commands is not placed on the history list. The **-h** flag causes the commands to be placed in the history list without executing them.

stop [**%***Job* ...]
Stops the current job or specified *Job* running in the background.

suspend
Stops the shell as if a **STOP** signal had been received.

switch (*string*)
Starts a **switch (***String***) case** *String* **:** ... **breaksw default:** ... **breaksw endsw** command sequence. This command sequence successively matches each case label against the value of the *String* variable. If none of the labels match before a default label is found, the execution begins after the default label.

time [*Command*]
The **time** command controls automatic timing of commands. If you do not specify the *Command* variable, the **time** command displays a summary of time used by this shell and its children. If you specify a command with the *Command* variable, it is timed. The shell then displays a time summary, as described under the **time** shell variable. If necessary, an extra shell is created to display the time statistic when the command completes.

The following example uses **time** with the **sleep** command:

```
time sleep
```

The output from this command looks similar to the following:

```
0.0u 0.0s 0:00 100% 44+4k 0+0io 0pf+0w
```

The output fields are as follows:

Field	**Description**
First	Number of seconds of CPU time devoted to the user process
Second	Number of seconds of CPU time consumed by the kernel on behalf of the user process
Third	Elapsed (wall clock) time for the command
Fourth	Total user CPU Time plus system time, as a percentage of elapsed time
Fifth	Average amount of shared memory used, plus average amount of unshared data space used, in kilobytes
Sixth	Number of block input and output operations
Seventh	Page faults plus number of swaps

umask [*Value*]
Determines file permissions. This *Value*, along with the permissions of the creating process, determines a file's permissions when the file is created. The default is 022. The current setting will be displayed if no *Value* is specified.

unalias *|*Pattern*
Discards all aliases with names that match the *Pattern* variable. All aliases are removed by the **unalias *** command. The absence of aliases does not cause an error.

unhash
Disables the use of the internal hash table to locate running programs.

unlimit [**-h**][*Resource*]
Removes the limitation on the *Resource* variable. If no *Resource* variable is specified, all resource limitations are removed. See the description of the **limit** command for the list of *Resource* names.

The **-h** flag removes corresponding hard limits. Only a user with root user authority can change hard limits.

unset ***|***_Pattern_	Removes all variables with names that match the _Pattern_ variable. Use **unset *** to remove all variables. If no variables are set, it does not cause an error.
unsetenv _Pattern_	Removes all variables from the environment whose name matches the specified _Pattern_. (See the **setenv** built-in command.)
wait	Waits for all background jobs. If the shell is interactive, an INTERRUPT (usually the Ctrl-C key sequence) disrupts the wait. The shell then displays the names and job numbers of all jobs known to be outstanding.
while (_Expression_) _Command_. . . **end**	Evaluates the _Commands_ between the **while** and the matching **end** statements while the expression specified by the _Expression_ variable evaluates nonzero. You can use the **break** statement to end and the **continue** statement to continue the loop prematurely. The **while** and **end** statements must appear alone on their input lines. If the input is from a terminal, prompts occur after the while (_Expression_) similar to the **foreach** statement.
@ [_Name_[_n_] **=** _Expression_]	Displays the values of all the shell variables when used with no arguments. Otherwise, sets the name specified by the _Name_ variable to the value of the _Expression_ variable. If the expression contains <, >, &, or | characters, this part of the expression must be placed within parentheses. When _n_ is specified, the _n_th component of the _Name_ variable is set to the _Expression_ variable. Both the _Name_ variable and its _n_th component must already exist. C language operators, such as *= and +=, are available. The space separating the _Name_ variable from the assignment operator is optional. Spaces are, however, required in separating components of the _Expression_ variable, which would otherwise be read as a single word. Special suffix operators, double plus sign (++) and double hyphen (- -) increase and decrease, respectively, the value of the _Name_ variable.

C Shell Expressions and Operators

The **@** built-in command and the **exit**, **if**, and **while** statements accept expressions that include operators similar to those of C language, with the same precedence. The following operators are available:

Operator	What it Means
`()`	change precedence
`~`	complement
`!`	negation
`*/ %`	multiply, divide, modulo
`+ -`	add, subtract
`<< > >`	left shift, right shift
`<= >= < >`	relational operators
`== != =~ !~`	string comparison/pattern matching
`&`	bitwise AND
`^`	bitwise exclusive OR
`|`	bitwise inclusive OR
`&&`	logical AND
`||`	logical OR

In the previous list, precedence of the operators decreases down the list (left to right, top to bottom).

Note: The operators + and - are right-associative. For example, evaluation of a + b - c is performed as follows:

```
a + (b - c)
```

and not as follows:

```
(a + b) - c
```

The ==, !=, =~, and !~ operators compare their arguments as strings; all others operate on numbers. The =~ and !~ operators are similar to == and !=, except that the rightmost side is a *pattern* against which the leftmost operand is matched. This reduces the need for use of the **switch** statement in shell procedures.

The logical operators or (||) and and (&&) are also available. They can be used to check for a range of numbers, as in the following example:

```
if ($#argv > 2 && $#argv < 7) then
```

In the preceding example, the number of arguments must be greater than 2 and less than 7.

Strings beginning with zero (0) are considered octal numbers. Null or missing arguments are considered 0. All expressions result in strings representing decimal numbers. Note that two components of an expression can appear in the same word. Except when next to components of expressions that are syntactically significant to the parser (& | < > ( )), expression components should be surrounded by spaces.

Also available in expressions as primitive operands are command executions enclosed in ( ) and file inquiries of the form (**-operator** *Filename*), where **operator** is one of the following:

r	Read access
w	Write access
x	Execute access
e	Existence
o	Ownership
z	Zero size
f	Plain file
d	Directory

The specified *Filename* is command and file-name expanded and then tested to see if it has the specified relationship to the real user. If *Filename* does not exist or is inaccessible, all inquiries return false(0). If the command runs successfully, the inquiry returns a value of true(1). Otherwise, if the command fails, the inquiry returns a value of false(0). If more detailed status information is required, run the command outside an expression and then examine the **status** shell variable.

Command Substitution in the C Shell

In *command substitution*, the shell executes a specified command and replaces that command with its output. To perform command substitution in the C shell, enclose the command or command string in backquotes (` `). The shell normally breaks the output from the command into separate words at blanks, tabs, and newline characters. It then replaces the original command with this output.

In the following example, the backquotes (` `) around the **date** command indicate that the output of the command will be substituted:

```
echo The current date and time is: `date`
```

The output from this command might look like:

```
The current date and time is: Wed Apr 8 13:52:14 CDT 1992
```

The C shell performs command substitution selectively on the arguments of built-in shell commands. This means that it does not expand those parts of expressions that are not evaluated. For commands that are not built-in, the shell substitutes the command name separately from the argument list. The substitution occurs in a child of the main shell, only after the shell performs input or output redirection.

If a command string is surrounded by " ", the shell treats only newline characters as word separators, thus preserving blanks and tabs within the word. In all cases, the single final newline character does not force a new word.

Nonbuilt-in C Shell Command Execution

When the C shell determines that a command is not a built-in shell command, it attempts to run the command with the **execv** subroutine. Each word in the **path** shell variable names a directory from which the shell attempts to run the command. If given neither the **-c** nor **-t** flag, the shell hashes the names in these directories into an internal table. The shell tries to call the **exec** subroutine on a directory only if there is a possibility that the command resides there. If you turn off this mechanism with the **unhash** command or give the shell the **-c** or **-t** flag, the shell concatenates with the given command name to form a path name of a file. The shell also does this in any case for each directory component of the **path** variable that does not begin with a /. The shell then attempts to run the command.

Parenthesized commands always run in a subshell. For example:

```
(cd ; pwd) ; pwd
```

displays the home directory without changing the current directory location. However, the command:

```
cd ; pwd
```

changes the current directory location to the home directory. Parenthesized commands are most often used to prevent the **chdir** command from affecting the current shell.

If the file has execute permission, but is not an executable binary to the system, then the shell assumes it is a file containing shell commands and runs a new shell to read it.

If there is an alias for the shell, then the words of the alias are prefixed to the argument list to form the shell command. The first word of the alias should be the full path name of the shell.

History Substitution in the C Shell

History substitution lets you modify individual words from previous commands to create new commands. History substitution makes it easy to repeat commands, repeat the arguments of a previous command in the current command, or fix spelling mistakes in the previous command with little typing.

History substitutions begin with the ! character and can appear anywhere on the command line, provided they do not nest (in other words, a history substitution cannot contain another history substitution). You can precede the ! with a \ to cancel the exclamation point's special meaning. In addition, if you place the ! before a blank, tab, newline character, =, or (, history substitution does not occur.

History substitutions also occur when you begin an input line with a ^. The shell echoes any input line containing history substitutions at the workstation before it executes that line.

This section discusses the following:
- "History Lists"
- "Event Specification" on page 210
- "Quoting with Single and Double Quotes" on page 211

History Lists

The history list saves commands that the shell reads from the command line that consist of one or more words. History substitution reintroduces sequences of words from these saved commands into the input stream.

The **history** shell variable controls the size of the history list. You must set the **history** shell variable either in the **.cshrc** file or on the command line with the built-in **set** command. The previous command is always retained regardless of the value of the **history** variable. Commands in the history list are numbered sequentially, beginning with 1. The built-in **history** command produces output similar to the following:

```
9 write michael
10 ed write.c
11 cat oldwrite.c
12 diff *write.c
```

The shell displays the command strings with their event numbers. The event number appears to the left of the command and represent when the command was entered in relation to the other commands in the history. It is not usually necessary to use event numbers to refer to events, but you can have the current event number displayed as part of your system prompt by placing an ! in the prompt string assigned to the **PROMPT** environment variable.

A full history reference contains an event specification, a word designator, and one or more modifiers in the following general format:

```
Event[.]Word:Modifier[:Modifier] . . .
```

Note: Only one word can be modified. A string that contains blanks is not allowed.

In the previous sample of **history** command output, the current event number is 13. Using this example, the following refer to previous events:

`!10`	Event number 10.
`!-2`	Event number 11 (the current event minus 2).
`!d`	Command word beginning with `d` (event number 12).
`!?mic?`	Command word containing the string `mic` (event number 9).

These forms, without further modification, simply reintroduce the words of the specified events, each separated by a single blank. As a special case, !! refers to the previous command; the command **!!** alone on an input line reruns the previous command.

Event Specification

To select words from an event, follow the event specification with a : and one of the following word designators (the words of an input line are numbered sequentially starting from 0):

0	First word (the command name)
n	*n*th argument
^	First argument
$	Last argument
%	Word matched by an immediately preceding ?*string*? search
x-y	Range of words from the *x*th word to the *y*th word
-y	Range of words from the first word (0) to the *y*th word
*****	First through the last argument, or nothing if there is only one word (the command name) in the event
x*	*x*th argument through the last argument
***x*-**	Same as *x** but omitting the last argument

If the word designator begins with a ^, $, *, -, or %, you can omit the colon that separates the event specification from the word designator. You can also place a sequence of the following modifiers after the optional word designator, each preceded by a colon:

h	Removes a trailing path name extension, leaving the head.
r	Removes a trailing `.xxx` component, leaving the root name.
e	Removes all but the `.xxx` trailing extension.
s/*OldWord***/***NewWord***/**	Substitutes the value of the *NewWord* variable for the value of the *OldWord* variable.

The left side of a substitution is not a pattern in the sense of a string recognized by an editor; rather, it is a word, a single unit without blanks. Normally, a / delimits the original word (*OldWord*) and its replacement (*NewWord*). However, you can use any character as the delimiter. In the following example, using the `%` as a delimiter allows a / to be included in the words:

```
s%/home/myfile%/home/yourfile%
```

The shell replaces an `&` with the *OldWord* text in the *NewWord* variable. In the following example, `/home/myfile` becomes `/temp/home/myfile`.

```
s%/home/myfile%/temp&%
```

The shell replaces a null word in a substitution with either the last substitution or with the last string used in the contextual scan `!?String?`. You can omit the trailing delimiter (/) if a newline character follows immediately. Use the following modifiers to delimit the history list:

t	Removes all leading path name components, leaving the tail
&	Repeats the previous substitution
g	Applies the change globally; that is, all occurrences for each line
p	Displays the new command, but does not run it
q	Quotes the substituted words, thus preventing further substitutions
x	Acts like the **q** modifier, but breaks into words at blanks, tabs, and new-line characters

When using the preceeding modifiers, the change applies only to the first modifiable word unless the **g** modifier precedes the selected modifier.

If you give a history reference without an event specification (for example, `!$`), the shell uses the previous command as the event. If a previous history reference occurs on the same line, the shell repeats the previous reference. Thus, the following sequence gives the first and last arguments of the command that matches `?foo?`.

```
!?foo?^ !$
```

A special abbreviation of a history reference occurs when the first nonblank character of an input line is a ^. This is equivalent to `!:s^` , thus providing a convenient shorthand for substitutions on the text of the previous line. The command `^ lb^ lib` corrects the spelling of `lib` in the command.

If necessary, you can enclose a history substitution in { } to insulate it from the characters that follow. For example, if you want to use a reference to the command:

```
ls -ld ~paul
```

to perform the command:

```
ls -ld ~paula
```

use the following construction:

```
!{l}a
```

In this example, `!{l}a` looks for a command starting with `l` and appends `a` to the end.

Quoting with Single and Double Quotes

To prevent further interpretation of all or some of the substitutions, enclose strings in single and double quotation marks. Enclosing strings in ' ' prevents further interpretation, while enclosing strings in ″ ″ allows further expansion. In both cases, the text that results becomes all or part of a single word.

Alias Substitution in the C Shell

An *alias* is a name assigned to a command or command string. The C shell allows you to assign aliases and use them as you would commands. The shell maintains a list of the aliases that you define.

After the shell scans the command line, it divides the commands into distinct words and checks the first word of each command, left to right, to see if there is an alias. If an alias is found, the shell uses the history mechanism to replace the text of the alias with the text of the command referenced by the alias. The resulting words replace the command and argument list. If no reference is made to the history list, the argument list is left unchanged.

For information about the C shell history mechanism, see "History Substitution in the C Shell" on page 209.

The **alias** and **unalias** built-in commands establish, display, and modify the alias list. Use the alias command in the following format:

```
alias [Name [WordList]]
```

The optional *Name* variable specifies the alias for the specified name. If you specify a word list with the *WordList* variable, the command assigns it as the alias of the *Name* variable. If you run the **alias** command without either optional variable, it displays all C shell aliases.

If the alias for the **ls** command is `ls -l`, the following command:

```
ls /usr
```

is replaced by the command:

```
ls -l /usr
```

The argument list is undisturbed because there is no reference to the history list in the command with an alias. Similarly, if the alias for the **lookup** command is as follows:

```
grep \!^ /etc/passwd
```

then the shell replaces `lookup bill` with the following:

```
grep bill /etc/passwd
```

In this example, `!^` refers to the history list, and the shell replaces it with the first argument in the input line, in this case `bill`.

You can use special pattern-matching characters in an alias. The following command:

```
alias lprint 'pr &bslash2.!* >

> print'
```

creates a command that formats its arguments to the line printer. The `!` character is protected from the shell in the alias by use of single quotation marks so that the alias is not expanded until the **pr** command runs.

If the shell locates an alias, it performs the word transformation of the input text and begins the alias process again on the reformed input line. If the first word of the next text is the same as the old, looping is prevented by flagging the alias to terminate the alias process. Other subsequent loops are detected and result in an error.

Variable and File-Name Substitution in the C Shell

The C Shell permits you to do variable and file-name substitutions.

This section discusses the following:

- “Variable Substitution in the C Shell”
- “File-Name Substitution in the C Shell” on page 214
- “File-Name Expansion” on page 214
- “File-Name Abbreviation” on page 215
- “Character Classes” on page 216
- “C Shell” on page 221

Variable Substitution in the C Shell

The C shell maintains a set of variables, each of which has as its value a list of zero or more words. Some of these variables are set by the shell or referred to by it. For instance, the **argv** variable is an image of the shell variable list, and words that comprise the value of this variable are referred to in special ways.

To change and display the values of variables, use the **set** and **unset** commands. Of the variables referred to by the shell, a number are toggles (variables that turn something on and off). The shell does not examine toggles for a value, only for whether they are set or unset. For instance, the **verbose** shell variable is a toggle that causes command input to be echoed. The setting of this variable results from issuing the **-v** flag on the command line.

Other operations treat variables numerically. The **@** command performs numeric calculations and the result is assigned to a variable. Variable values are, however, always represented as (zero or more) strings. For numeric operations, the null string is considered to be zero, and the second and subsequent words of multiword values are ignored.

When you issue a command, the shell parses the input line and performs alias substitution. Next, before running the command, it performs variable substitution. The $ character keys the substitution. It is, however, passed unchanged if followed by a blank, tab, or newline character. Preceding the $ character with a \ prevents this expansion, except in two cases:

- The command is enclosed in " ". In this case, the shell always performs the substitution.
- The command is enclosed in ' '. In this case, the shell never performs the substitution. Strings enclosed in ' ' are interpreted for command substitution. (See “Command Substitution in the C Shell” on page 208.)

The shell recognizes input and output redirection before variable expansion, and expands each separately. Otherwise, the command name and complete argument list expands together. It is therefore possible for the first (command) word to generate more than one word, the first of which becomes the command name and the rest of which become parameters.

Unless enclosed in " " or given the **:q** modifier, the results of variable substitution might eventually be subject to command and file-name substitution. When enclosed by double quotation marks, a variable with a value that consists of multiple words expands to a single word or a portion of a single word, with the words of the variable’s value separated by blanks. When you apply the **:q** modifier to a substitution, the variable expands to multiple words. Each word is separated by a blank and enclosed in double quotation marks to prevent later command or file-name substitution.

The following notations allow you to introduce variable values into the shell input. Except as noted, it is an error to reference a variable that is not set with the **set** command.

You can apply the modifiers **:gh**, **:gt**, **:gr**, **:h**, **:r**, **:q**, and **:x** to the following substitutions. If { } appear in the command form, then the modifiers must be placed within the braces. Only one **:** modifier is permitted on each variable expansion.

$*Name*	
${*Name***}**	Replaced by the words assigned to the *Name* variable, each separated by a blank. Braces insulate the *Name* variable from any following characters that would otherwise be part of it. Shell variable names start with a letter and consist of up to 20 letters and digits, including the underline (_) character. If the *Name* variable does not specify a shell variable but is set in the environment, then its value is returned. The modifiers preceded by colons, as well as the other forms described here, are not available in this case.
$*Name***[***number***]**	
${*Name***[***number***]}**	Selects only some of the words from the value of the *Name* variable. The number is subjected to variable substitution and might consist of a single number, or two numbers separated by a -. The first word of a variable's string value is numbered 1. If the first number of a range is omitted, it defaults to 1. If the last number of a range is omitted, it defaults to **$#***Name*. The * symbol selects all words. It is not an error for a range to be empty if the second argument is omitted or is in a range.
$#*Name*	
${#*Name***}**	Gives the number of words in the *Name* variable. This can be used in a **[***number***]** as shown above. For example, `$Name[$#Name]`**.**
$0	Substitutes the name of the file from which command input is being read. An error occurs if the name is not known.
$*number*	
${*number***}**	Equivalent to `$argv[number]`.
$*	Equivalent to `$argv[*]`.

The following substitutions may not be changed with **:** modifiers:

$?*name*	
${?*name***}**	Substitutes the string 1 if the *name* variable is set, zero (0) if this variable is not set.
$?0	Substitutes 1 if the current input file name is known, zero (0) if the file name is not known.
$$	Substitutes the (decimal) process number of the parent shell.
$<	Substitutes a line from standard input, without further interpretation. Use this substitution to read from the keyboard in a shell procedure.

File-Name Substitution in the C Shell

The C shell provides several shortcuts to save time and keystrokes. If a word contains any of the characters *, ?, [ ], or { }, or begins with a tilde (~), that word is a candidate for file-name substitution. The C shell regards the word as a pattern and replaces the word with an alphabetized list of file names matching the pattern.

The current collating sequence is used, as specified by the **LC_COLLATE** or **LANG** environment variables. In a list of words specifying file-name substitution, an error results if no patterns match an existing file name. However, it is not required that every pattern match. Only the character-matching symbols *, ?, and [ ] indicate pattern-matching or file-name expansion. The tilde (~) and { } characters indicate file-name abbreviation.

File-Name Expansion

The * character matches any string of characters, including the null string. For example, in a directory containing the files:

```
a aa aax alice b bb c cc
```

the command `echo a*` prints all files names beginning with the character `a`:

```
a aa aax alice
```

> **Note:** When file names are matched, the characters dot (.) and / must be matched explicitly.

The ? character matches any single character. The following command:

```
ls a?x
```

lists every file name beginning with the letter a, followed by a single character, and ending with the letter x:

```
aax
```

To match a single character or a range of characters, enclose the character or characters inside of [ ]. The following command:

```
ls [abc]
```

lists all file names exactly matching one of the enclosed characters:

```
a b c
```

Within brackets, a lexical range of characters is indicated by `[a-z]`. The characters matching this pattern are defined by the current collating sequence.

File-Name Abbreviation

The tilde (~) and { characters indicate file-name abbreviation. A ~ at the beginning of a file name is used to represent home directories. Standing alone, the ~ character expands to your home directory as reflected in the value of the **home** shell variable. For example, the following command:

```
ls ~
```

lists all files and directories located in your **$HOME** directory.

When the command is followed by a name consisting of letters, digits, and - characters, the shell searches for a user with that name and substitutes that user's **$HOME** directory.

> **Note:** If the ~ character is followed by a character other than a letter or /, or appears anywhere except at the beginning of a word, it does not expand.

To match characters in file names without typing the entire file name, use { } around the file names. The pattern `a{b,c,d}e` is another way of writing `abe ace ade`. The shell preserves the left-to-right order and separately stores the results of matches at a low level to preserve this order. This construct might be nested. Thus, the following:

```
~source/s1/{oldls,ls}.c
```

expands to:

```
/usr/source/s1/oldls.c /usr/source/s1/ls.c
```

if the home directory for **source** is **/usr/source**. Similarly, the following:

```
../{memo,*box}
```

might expand to:

```
../memo ../box ../mbox
```

> **Note:** `memo` is not sorted with the results of matching `*box`. As a special case, the {, }, and { } characters are passed undisturbed.

Character Classes

You can also use character classes to match file names within a range indication. The following format instructs the system to match any single character belonging to the specified class:

```
[:charclass:]
```

The following classes correspond to **ctype** subroutines:

Character Class	**Definition**
alnum	Alphanumeric characters
alpha	Uppercase and lowercase letters
cntrl	Control characters
digit	Digits
graph	Graphic characters
lower	Lowercase letters
print	Printable characters
punct	Punctuation character
space	Space, horizontal tab, carriage return, newline, vertical tab, or form-feed character
upper	Uppercase characters
xdigit	Hexadecimal digits

Suppose that you are in a directory containing the following files:

```
a aa aax Alice b bb c cc
```

Type the following command at a C shell prompt:

```
    ls [:lower:]
```

Press Enter.

The C shell lists all file names that begin with lowercase characters:

```
a aa aax b bb c cc
```

For more information about character class expressions, refer to the **ed** command.

Environment Variables in the C Shell

Certain variables have special meaning to the C shell. Of these, **argv**, **cwd**, **home**, **path**, **prompt**, **shell**, and **status** are always set by the shell. Except for the **cwd** and **status** variables, this action occurs only at initialization. These variables maintain their settings unless you explicitly reset them.

The **csh** command copies the **USER**, **TERM**, **HOME**, and **PATH** environment variables into the **csh** variables, **user**, **term**, **home**, and **path**, respectively. The values are copied back into the environment whenever the normal shell variables are reset. The **path** variable cannot be set in other than in the **.cshrc** file, because **csh** subprocesses import the path definition from the environment and reexport it if changed.

The following variables have special meanings:

argv	Contains the arguments passed to shell scripts. Positional parameters are substituted from this variable.
cdpath	Specifies a list of alternate directories to be searched by the **chdir** or **cd** command to find subdirectories.
cwd	Specifies the full path name of the current directory.

echo	Set when the **-x** command line flag is used; when set, causes each command and its arguments to echo just before being run. For commands that are not built-in, all expansions occur before echoing. Built-in commands are echoed before command and file-name substitution because these substitutions are then done selectively.
histchars	Specifies a string value to change the characters used in history substitution. Use the first character of its value as the history substitution character, this replaces the default character, !. The second character of its value replaces the ^ character in quick substitutions. **Note:** Setting the **histchars** value to a character used in command or file names might cause unintentional history substitution.
history	Contains a numeric value to control the size of the history list. Any command that is referenced within the number of events permitted is not discarded. Very large values of the **history** variable might cause the shell to run out of memory. Regardless of whether this variable is set, the C shell always saves the last command that ran on the history list.
home	Indicates your home directory, initialized from the environment. The file-name expansion of the tilde (~) character refers to this variable.
ignoreeof	Specifies that the shell ignore an end-of-file character from input devices that are workstations. This prevents shells from accidentally being killed when the shell reads an end-of-file character (Ctrl-D).
mail	Specifies the files where the shell checks for mail. This is done after each command completion which results in a prompt if a specified time interval has elapsed. The shell displays the message `Mail in file.` if the file exists with an access time less than its change time. If the first word of the value of the **mail** variable is numeric, it specifies a different mail-checking time interval (in seconds); the default is 600 (10 minutes). If you specify multiple mail files, the shell displays the message `New mail in file`, when there is mail in the specified file.
noclobber	Places restrictions on output redirection to ensure that files are not accidentally destroyed and that redirections append to existing files.
noglob	Inhibits file-name expansion. This is most useful in shell scripts that do not deal with file names, or when a list of file names has been obtained and further expansions are not desirable.
nonomatch	Specifies that no error results if a file name expansion does not match any existing files; rather, the primitive pattern returns. It is still an error for the primitive pattern to be malformed.
notify	Specifies that the shell send asynchronous notification of changes in job status. The default presents status changes just before displaying the shell prompt.
path	Specifies directories in which commands are sought for execution. A null word specifies the current directory. If there is no **path** variable set, then only full path names can run. The default search path (from the **/etc/environment** file used during login) is as follows: `/usr/bin /etc /usr/sbin /usr/ucb /usr/bin/X11 /sbin` A shell given neither the **-c** nor the **-t** flag normally hashes the contents of the directories in the **path** variable after reading the **.cshrc** and also each time the **path** variable is reset. If new commands are added to these directories while the shell is active, you must give the **rehash** command. Otherwise, the commands might not be found.
prompt	Specifies the string displayed before each command is read from an interactive workstation input. If an ! appears in the string, it is replaced by the current event number. If the ! character is in a quoted string enclosed by single or double quotation marks, the ! character must be preceded by a \. The default prompt for users without root authority is % . The default prompt for the user with root authority is `#`.
savehist	Specifies a numeric value to control the number of entries of the history list that are saved in the **~/.history** file when you log out. Any command referenced in this number of events is saved. During startup, the shell reads **~/.history** into the history list, enabling history to be saved across logins. Very large values of the **savehist** variable slow down the shell startup.
shell	Specifies the file in which the C shell resides. This is used in forking shells to interpret files that have execute bits set, but which are not executable by the system. This is initialized to the home of the C shell.
status	Specifies the status returned by the last command. If the command ends abnormally, 0200 is added to the status. Built-in commands that are unsuccessful return an exit status of 1. Successful built-in commands set status to a value of 0.
time	Controls automatic timing of commands. If this variable is set, any command that takes more than the specified number of CPU seconds will display a line of resources used, at the end of execution. For more information about the default outputs, see the built-in **time** command.

verbose	Set by the **-v** command line flag, this variable causes the words of each command to display after history substitution.

Input and Output Redirection in the C Shell

Before the C shell executes a command, it scans the command line for redirection characters. These special notations direct the shell to redirect input and output.

You can redirect the standard input and output of a command with the following syntax statements:

< *File*	Opens the specified *File* (which is first variable, command, and file name expanded) as the standard input.
<<*Word*	Reads the shell input up to the line that matches the value of the *Word* variable. The *Word* variable is not subjected to variable, file name, or command substitution. Each input line is compared to the *Word* variable before any substitutions are done on the line. Unless a quoting character (\, ″, ' or `) appears in the *Word* variable, the shell performs variable and command substitution on the intervening lines, allowing the \ character to quote the $, \, and ` characters. Commands that are substituted have all blanks, tabs, and newline characters preserved, except for the final newline character, which is dropped. The resultant text is placed in an anonymous temporary file, which is given to the command as standard input.

> *File*

>!*File*

>& *File*

>&! *File* Uses the specified *File* as standard output. If *File* does not exist, it is created. If *File* exists, it is truncated, and its previous contents are lost. If the **noclobber** shell variable is set, *File* must not exist or be a character special file, or an error results. This helps prevent accidental destruction of files. In this case, use the forms including an ! to suppress this check. *File* is expanded in the same way as < input file names. The form >& redirects both standard output and standard error to the specified *File*. The following example shows how to separately redirect standard output to **/dev/tty** and standard error to **/dev/null**. The parentheses are required to allow standard output and standard error to be separate.

```
% (find / -name vi -print > /dev/tty) >& /dev/null
```

> >*File*

> >! *File*

> >& *File*

> >&! *File* Uses the specified *File* as standard output like >, but *appends* output to the end of *File*. If the **noclobber** shell variable is set, an error results if *File* does not exist, unless one of the forms including an ! is given. Otherwise, it is similar to >.

A command receives the environment in which the shell was invoked, as changed by the input/output parameters and the presence of the command as a pipeline. Thus, unlike some previous shells, commands that run from a shell script do not have access to the text of the commands by default. Rather, they receive the original standard input of the shell. Use the << mechanism to present inline data, which allows shell command files to function as components of pipelines and also lets the shell block read its input. Note that the default standard input for a command run detached is not changed to the empty **/dev/null** file. Rather, the standard input remains the original standard input of the shell.

To redirect the standard error through a pipe with the standard output, use the form |& rather than only the |.

Control Flow

The shell contains commands that can be used to regulate the flow of control in command files (shell scripts) and (in limited but useful ways) from shell command-line input. These commands all operate by forcing the shell to repeat, or skip, in its input.

The **foreach, switch**, and **while** statements, and the **if-then-else** form of the **if** statement, require that the major keywords appear in a single simple command on an input line.

If the shell input is not searchable, the shell buffers input whenever a loop is being read and searches the internal buffer to do the rereading implied by the loop. To the extent that this is allowed, backward **goto**s succeed on inputs that you cannot search.

Job Control in the C Shell

The shell associates a job number with each process. The shell keeps a table of current jobs and assigns them small integer numbers. When you start a job in the background with an **&** , the shell prints a line that looks like the following:

```
[1] 1234
```

This line indicates that the job number is `1` and that the job is composed of a single process with a process ID of `1234`. Use the built-in **jobs** command to see the table of current jobs.

A job running in the background competes for input if it tries to read from the workstation. Background jobs can also produce output for the workstation that gets interleaved with the output of other jobs.

You can refer to jobs in the shell in several ways. Use the % character to introduce a job name. This name can be either the job number or the command name that started the job, if this name is unique. For example, if a **make** process is running as job 1, you can refer to it as `%1`. You can also refer to it as `%make`, if there is only one suspended job with a name that begins with the string `make`. You can also use the following:

```
%?String
```

to specify a job whose name contains the `String` variable, if there is only one such job.

The shell detects immediately whenever a process changes its state. If a job becomes blocked so that further progress is impossible, the shell sends a message to the workstation. This message displays only after you press the Enter key. If, however, the **notify** shell variable is set, the shell immediately issues a message that indicates changes in the status of background jobs. Use the built-in **notify** command to mark a single process so that its status changes are promptly reported. By default, the **notify** command marks the current process.

List of C Shell Built-in Commands

@	Displays the value of specified shell variables.
alias	Displays specified aliases or all aliases.
bg	Puts the current or specified jobs into the background.
break	Resumes running after the end of the nearest enclosing **foreach** or **while** command.
breaksw	Breaks from a **switch** command.
case	Defines a label in a **switch** command.
cd	Changes the current directory to the specified directory.
chdir	Changes the current directory to the specified directory.

continue	Continues execution of the nearest enclosing **foreach** or **while** command.
default	Labels the default case in a **switch** statement.
dirs	Displays the directory stack.
echo	Writes character strings to the standard output of the shell.
else	Runs the commands that follow the second **else** in an **if (***Expression***) then** ...**else if (***Expression2***) then** ... **else** ... **endif** command sequence.
end	Signifies the end of a sequence of commands preceded by the **foreach** command.
endif	Runs the commands that follow the second **then** statement in an **if (***Expression***) then** ... **else if (***Expression2***) then** ... **else** ... **endif** command sequence.
endsw	Marks the end of a **switch (***String***) case** *String* **:** ... **breaksw default:** ... **breaksw endsw** command sequence. This command sequence successively matches each case label against the value of the *String* variable. Execution continues after the **endsw** command if a **breaksw** command is executed or if no label matches and there is no default.
eval	Reads variable values as input to the shell and executes the resulting command or commands in the context of the current shell.
exec	Runs the specified command in place of the current shell.
exit	Exits the shell with either the value of the status shell variable or the value of the specified expression.
fg	Brings the current or specified jobs into the foreground, continuing them if they are stopped.
foreach	Successively sets a *Name* variable for each member specified by the *List* variable and a sequence of commands, until reaching an **end** command.
glob	Displays list using history, variable, and file-name expansion.
goto	Continues to run after a specified line.
hashstat	Displays statistics indicating how successful the hash table has been at locating commands.
history	Displays the history event list.
if	Runs a specified command if a specified expression is true.
jobs	Lists the active jobs.
kill	Sends either the **TERM** (terminate) signal or the signal specified by the *Signal* variable to the specified job or process.
limit	Limits usage of a specified resource by the current process and each process it creates.
login	Ends a login shell and replaces it with an instance of the **/usr/sbin/login** command.
logout	Ends a login shell.
nice	Sets the priority of commands run in the shell.
nohup	Causes hangups to be ignored for the remainder of a procedure.
notify	Causes the shell to notify you asynchronously when the status of the current or a specified job changes.
onintr	Controls the action of the shell on interrupts.
popd	Pops the directory stack and returns to the new top directory.
pushd	Exchanges elements of the directory stack.
rehash	Causes recomputation of the internal hash table containing the contents of the directories in the path shell variable.
repeat	Runs the specified command, subject to the same restrictions as the **if** command, the number of times specified.
set	Shows the value of all shell variables.
setenv	Modifies the value of the specified environment variable.
shift	Shifts the specified variable to the left.
source	Reads command specified by the *Name* variable.
stop	Stops the current or specified jobs running in the background.
suspend	Stops the shell as if a **STOP** signal has been received.
switch	Starts a **switch (***String***) case** *String* **:** ... **breaksw default:** ... **breaksw endsw** command sequence. This command sequence successively matches each case label against the value of the *String* variable. If none of the labels match before a default label is found, the execution begins after the default label.
time	Displays a summary of the time used by the shell and its child processes.
umask	Determines file permissions.
unalias	Discards all aliases with names that match the *Pattern* variable.
unhash	Disables the use of the internal hash table to locate running programs.

unlimit	Removes resource limitations.
unset	Removes all variables having names that match the *Pattern* variable.
unsetenv	Removes all variables from the environment whose names match the specified *Pattern* variable.
wait	Waits for all background jobs.
while	Evaluates the commands between the **while** and the matching **end** command sequence while an expression specified by the *Expression* variable evaluates nonzero.

Related Information

Korn Shell

The **ksh** and **stty** commands.

The **alias**, **cd**, **export**, **fc**, **getopts**, **read**, **set**, and **typeset** Korn shell commands.

The **/etc/passwd** file.

Bourne Shell

The **bsh** or **Rsh** command, **login** command.

The Bourne shell **read** special command.

The **setuid** subroutine, **setgid** subroutine.

The **null** special file.

The **environment** file, **profile** file format.

C Shell

The **csh** command, **ed** command.

The **alias**, **unalias**, **jobs**, **notify** and **set** C Shell built-in commands.

Chapter 13. AIX Documentation

This section discusses the online documentation available for AIX.

IBM eServer pSeries Information Center

IBM eServer pSeries Information Center provides a portal to AIX documentation as well as access to tools and resources. From the Information Center, the entire AIX software documentation library for releases 4.3, 5.1, and 5.2 are available. Each book released in 5.1 and 5.2 are available in PDF format, and abstracts for books released in 5.2 are provided. Other tools and resources include:

- An error message database showing users what the error messages mean and, in many cases, how they can recover from them. This database also provides information for LED codes, and error identifiers.
- A resources page that links users to other IBM and non-IBM Web sites proven useful to system administrators, application developers, and users.
- Several how-to's that provide users with step-by-step instructions for completing system administrator and user tasks.
- Several FAQs (frequently asked questions) that provide users with quick answers to common questions.
- Links to related frequently used documentation from IBM, including white papers, Redbooks, and technical reports on topics such as RS/6000 SP and HACMP for AIX.
- A link to the documentation search tool is provided for each release, along with links to the release notes and readme files.
- A link to the entire pSeries and RS/6000 hardware documentation library.

Using the IBM eServer pSeries Information Center

To bring up the Information Center on an AIX system, do the following:

- On the AIX command line, type:

  ```
  infocenter
  ```

  Press Enter.

  OR
- From the CDE Help panel, select the Information Center icon.

  OR
- Go to the following Web address:

  http://publib16.boulder.ibm.com/pseries/en_US/infocenter/base

Documentation Library Service

The Documentation Library Service allows you to read, search, and print online HTML documents and provides a library application that appears in your web browser. The application includes links to read installed documents and a search form that you can use to search for text. When you search, a results page displays the results of the search with links to the documents containing the search target words.

Starting with AIX 5.1, you can also download printable versions of books.

Two types of forms are provided: a global search form that shows all the volumes installed on a search server, and a specific search form that only searches a specific set of volumes, such as the manuals for an application.

The documentation library service allows you to search documents that have been registered with the search service. The system administrator registers the documents. You cannot search the Internet or any unregistered documents on your search server.

If you write HTML documents at your site, your system administrator can add these documents to the documentation library so that you can read, search, and print the documents.

Using the Documentation Library Service

You can use the Documentation Library Service to do the following tasks:

- To read the documentation installed in your system's default library, do one of the following:
  - Type `docsearch` at the command line.
  - Open the CDE Desktop Help subpanel. Click on the documentation search service icon, which looks like binoculars.

    Note: If you have a copy of the AIX 4.3 (or later version) CD, you can read it on a PC. Insert the CD into the CD-ROM drive on your PC. If the documentation CD is AIX 4.3.3 or later, use a Web browser to open the CD file called **readme.htm** that is located in the top directory of the CD. If you have an AIX 4.3.0 through AIX 4.3.2 version of the Base Documentation CD, open the **usr/share/man/info/en_US/a_doc_lib/aixgen/topnav/topnav.htm** file.
- To open a library stored on a remote documentation server, in your browser's location bar, type the following Web address:

  ```
  http://server_name[:port_number]/cgi-bin/ds_form
  ```

  The global search form opens, where you can search the documents stored on the server with the name that you specified in *server_name*. You can also view all of the books by category.

  Note: You need type the *port_number* only if the port is not the standard 80.

  For example, if you want to search the registered documents on a search server named `hinson` and it uses port 80, type:

  ```
  http://hinson/cgi-bin/ds_form
  ```

  After the search form for a server displays in your browser, you can create a bookmark that takes you back to the server. Your system administrator can also create a Web page that contains links to all of the different documentation servers in an organization.

  The documentation is also available for reading and searching at the following Web address: http://www.ibm.com/servers/aix/library. Note that while this site contains the AIX base operating system documentation, it may not contain other documentation installed on your local documentation server.
- Specific search forms are usually launched from search links inside HTML documents. They typically appear on the pages of applications manuals or help files. For example, each page in the library has a search link. Clicking on one of these search links launches a specific search form that allows you to search only the library volumes.

After the library application opens, you can click the **Help** link in the upper-right corner for instructions about how to use the library.

Changing the Documentation Library Service Language

By default, when you open the CDE Desktop icons for the documentation search service or the base library, the documents display in the same language as your CDE Desktop.

However, you might need to see the documentation in a language that is different from your desktop language. For example, your desktop runs in your native language but the manuals may only be available in English. You can change your documentation language so that documents display in a different language from that used in your desktop.

Notes:

1. The procedure described below does not affect the language used if you are opening a document or search form from an HTML link inside another document. These steps affect only the language used for the documentation search service or the base library desktop icons.
2. Make sure there is documentation already installed for the language you want to use.

You can change your documentation language by running the following command:

```
/usr/bin/chdoclang locale
```

Where *locale* is the locale name that is the new language for viewing and searching documentation. Locale names can be found in "Locale Naming Conventions" in *AIX 5L Version 5.2 National Language Support Guide and Reference*.

You must log out and then log back in to see the language change take effect.

If you are using the CDE Desktop, you must also edit your Desktop file **$HOME/.dtprofile** so that your documentation language setting in your **$HOME/.profile** file will be read during CDE login. To do this, complete the following steps:

1. Open your **.dtprofile** file in the dtpad editor by typing the following command:
   ```
   dtpad $HOME/.dtprofile
   ```
2. Find the line that contains the following text:
   ```
   DTSOURCEPROFILE=true
   ```
3. If there are any comment (**#**) characters at the start of that line, delete only the **#** characters, not the entire line. If there are no comment characters, close the editor.
4. Save your changed **.dtprofile** file.
5. Log out and log back in.

For example, if you want to change your documentation language to Spanish (locale name es_ES), type the following command:

```
/usr/bin/chdoclang es_ES
```

Log out and log back in to your desktop.

After you change your documentation language, you can delete the language setting so that documentation will again display in the same language as your desktop. To delete your language setting, type the following command:

```
/usr/bin/chdoclang -d
```

Log out and log back in to your desktop.

Appendix. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Dept. LRAS/Bldg. 003
11400 Burnet Road
Austin, TX 78758-3498
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Trademarks

Index

Readers' Comments — We'd Like to Hear from You

AIX 5L Version 5.2
System User's Guide: Operating System and Devices

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name

Address

Company or Organization

Phone No.

Readers' Comments — We'd Like to Hear from You

Cut or Fold
Along Line

Fold and Tape

Please do not staple

Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Information Development
Department H6DS-905-6C006
11501 Burnet Road
Austin, TX
78758-3493

Fold and Tape

Please do not staple

Fold and Tape

Cut or Fold
Along Line