AIX 5L Version 5.1

General Programming Concepts: Writing and Debugging Programs

AIX 5L Version 5.1

General Programming Concepts: Writing and Debugging Programs

Fourth Edition (April 2001)

Before using the information in this book, read the general information in “Appendix B. Notices” on page 983.

This edition applies to AIX 5L Version 5.1 and to all subsequent releases of this product until otherwise indicated in new editions.

A reader’s comment form is provided at the back of this publication. If the form has been removed, address comments to Publications Department, Internal Zip 9561, 11400 Burnet Road, Austin, Texas 78758-3493. To send comments electronically, use this commercial Internet address: aix6kpub@austin.ibm.com. Any information that you supply may be used without incurring any obligation to you.

Contents

About This Book

This book introduces you to the programming tools and interfaces available for writing and debugging application programs using the AIX operating system.

Who Should Use This Book

This book is intended for programmers who write and debug application programs on the AIX operating system. Users of this book should be familiar with the C programming language and AIX usage (entering commands, creating and deleting files, editing files, and moving around in the file system).

Highlighting

The following highlighting conventions are used in this book:

Bold	Identifies commands, subroutines, keywords, files, structures, directories, and other items whose names are predefined by the system. Also identifies graphical objects such as buttons, labels, and icons that the user selects.
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.

ISO 9000

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

Related Publications

The following books contain information about or related to writing programs:

- *AIX 5L Version 5.1 Kernel Extensions and Device Support Programming Concepts*
- *AIX 5L Version 5.1 Communications Programming Concepts* (“About This Book”)
- *AIX 5L Version 5.1 AIXwindows Programming Guide*
- *AIX 5L Version 5.1 System Management Guide: Operating System and Devices*
- *AIX 5L Version 5.1 System Management Guide: Communications and Networks*
- *AIX 5L Version 5.1 Commands Reference*
- *AIX 5L for POWER-based Systems Keyboard Technical Reference*
- *AIX 5L Version 5.1 Technical Reference: Base Operating System and Extensions Volume 1*
- *AIX 5L Version 5.1 Technical Reference: Base Operating System and Extensions Volume 2*
- *Understanding the Diagnostic Subsystem for AIX*

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

- AIX
- IBM

Microsoft, MS-DOS, Windows, and WindowsNT are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be the trademarks or service marks of others.

Chapter 1. Tools and Utilities

This chapter provides an overview of the operating system tools and utilities that you can use to develop C language programs.

The AIX Operating System provides many tools to help you develop C language programs. To access these tools, you enter a command on the command line. The tools provide help in the following programming areas:

- "Entering a Program into the System"
- "Checking a Program"
- "Compiling and Linking a Program"
- "Correcting Errors in a Program" on page 2
- "Building and Maintaining a Program" on page 2

"Subroutines" on page 2 and "Shell Commands" on page 2 are provided for use in a C language program.

Entering a Program into the System

The system has a line editor called **ed** for use in entering a program into a file. The system also has the full-screen editor called **vi**, which displays one full screen of data at a time and allows interactive editing of a file.

Checking a Program

The following commands allow you to check the format of a program for consistency and accuracy:

lint Checks for syntax and data type errors in a C language source program. The **lint** command checks these areas of a program more carefully than the C language compiler does, and displays many messages that point out possible problems.

cb Reformats a C language source program into a consistent format that uses indentation levels to show the structure of the program.

cflow Generates a diagram of the logic flow of a C language source program.

cxref Generates a list of all external references for each module of a C language source program, including where the reference is resolved (if it is resolved in the program).

Compiling and Linking a Program

To make source code into a program that the system can run, you need to process the source file with a compiler program and a linkage editor.

A compiler is a program that reads program text from a file and changes the programming language in that file to a form that the system understands. The linkage editor connects program modules together and determines how to put the finished program into memory. To create this final form of the program, the system does the following:

- If a file contains compiler source code, the compiler translates it into object code.
- If a file contains assembler language, the assembler translates it into object code.
- The linkage editor links the object files created in the previous step with any other object files specified in the compiler command.

Other programming languages available for use on the operating system include the FORTRAN, Pascal, and Assembler languages. Refer to documentation on these programming languages for information on compiling and linking programs written in them.

You can write parts of a program in different languages and have one main routine call and start the separate routines to execute, or use the **cc** program to both assemble and link the program.

Correcting Errors in a Program

The following debugging tools are available for use:

- **dbx** symbolic debugger can be used to debug programs written in C language, Pascal, FORTRAN, and Assembler language. For more information, see “dbx Symbolic Debug Program Overview” on page 63.
- **adb** “adb Debug Program Overview” on page 31 debugger provides subcommands to examine, debug, and repair executable binary files and to examine non-ASCII data files.
- Kernel Debug Program can help to determine errors in code running in the kernel. The primary application of this debugger is debugging device drivers.

When syntax errors or parameter naming inconsistencies are discovered in a program file, a text editor or string-searching and string-editing programs can be used to locate and change strings in the file. String-searching and string-editing programs include the **grep**, **sed**, and **awk** commands. To make many changes in one or more program files, you can include the commands in a shell program and then run the shell program to locate and change the code in the files.

Building and Maintaining a Program

Two facilities are provided to help you control program changes and build a program from many source modules. These commands can be particularly useful in software development environments in which many source modules are produced.

The **make** command builds a program from source modules. Since the **make** command compiles only those modules changed since the last build, its use can reduce compilation time when many source modules must be processed.

“Chapter 23. Source Code Control System (SCCS)” on page 629 allows you to maintain separate versions of a program without storing separate, complete copies of each version. The use of SCCS can reduce storage requirements and help in tracking the development of a project that requires keeping many versions of large programs.

Subroutines

Subroutines from system libraries handle many complex or repetitive programming situations so that you can concentrate on unique programming situations. See Subroutines Overview (“Chapter 24. Subroutines, Example Programs, and Libraries” on page 635) for information on using subroutines and for lists of many of the subroutines available on the system.

Shell Commands

You can include the functions of many of the shell commands in a C language program. Any shell command used in a program must be available on all systems that use the program.

You can then use the **fork** and **exec** subroutines in a program to run the command as a process in a part of the system that is separate from the program. The **system** subroutine also runs a shell command in a program, and the **popen** subroutine uses shell filters.

Chapter 2. The Curses Library

The Curses library provides a set of functions that enable you to manipulate a terminal's display regardless of the terminal type. Throughout this documentation, the Curses library is referred to as curses.

The basis of curses programming is the window data structure. Using this structure, you can manipulate data on a terminal's display. You can instruct curses to treat the entire terminal display as one large window or you can create multiple windows on the display. The windows can be different sizes and can overlap one another. A typical curses application has a single large window and one subwindow inside.

Each window on a terminal's display has its own window data structure. This structure keeps state information about the window such as its size and where it is located on the display. Curses uses the window data structure to obtain relevant information it needs to carry out your instructions.

Terminology

When programming with curses, you should be familiar with the following terms:

Term	Definition
current character	The character that the logical cursor is currently on.
current line	The line that the logical cursor is currently on.
curscr	A virtual default window provided by curses. The curscr (current screen) is an internal representation of what currently appears on the terminal's external display. Do not modify the curscr.
display	A physical display connected to a workstation.
logical cursor	The cursor location within each window. The window data structure keeps track of the location of its logical cursor.
pad	A type of window that is larger than the dimensions of the terminal's display.
physical cursor	The cursor that appears on a display. The workstation uses this cursor to write to the display. There is only one physical cursor per display.
screen	The window that fills the entire display. The screen is synonymous with the stdscr.
stdscr	A virtual default window (standard screen) provided by curses that represents the entire display.
window	A pointer to a C data structure and the graphic representation of that data structure on the display. A window can be thought of as a two-dimensional array representing how all or part of the display looks at any point in time.

Naming Conventions

A single curses subroutine can have two or more versions. Curses subroutines with multiple versions follow distinct naming conventions that identify the separate versions. These conventions add a prefix to a standard curses subroutine and identify what arguments the subroutine requires or what actions take place when the subroutine is called. The different versions of curses subroutine names use three prefixes:

Prefix	Description
w	Identifies a subroutine that requires a window argument.
p	Identifies a subroutine that requires a pad argument.
mv	Identifies a subroutine that first performs a move to the program-supplied coordinates.

Some curses subroutines with multiple versions do not include one of the preceding prefixes. These subroutines use the curses default window stdscr (standard screen). The majority of subroutines that use the stdscr are macros created in the **/usr/include/curses.h** file using **#define** statements. The

preprocessor replaces these statements at compilation time. As a result, these macros do not appear in the compiled assembler code, a trace, a debugger, or the curses source code.

If a curses subroutine has only a single version, it does not necessarily use stdscr. For example, the **printw** subroutine prints a string to the stdscr. The **wprintw** subroutine prints a string to a specific window by supplying the *window* argument. The **mvprintw** subroutine moves the specified coordinates to the stdscr and then performs the same function as the **printw** subroutine. Likewise, the **mvwprintw** subroutine moves the specified coordinates to the specified window and then performs the same function as the **wprintw** subroutine.

Structure of a Curses Program

In general, a curses program has the following progression:

- Start curses.
- Check for color support (optional).
- Start color (optional).
- Create one or more windows.
- Manipulate windows.
- Destroy one or more windows.
- Stop curses.

Your program does not have to follow this progression exactly.

Return Values

With a few exceptions, all curses subroutines return either the integer value ERR or the integer value OK. Subroutines that do not follow this convention are noted appropriately. Subroutines that return pointers always return a null pointer on an error.

Initializing Curses

initscr	Initializes the curses subroutine library and its data structures
newterm	Sets up a new terminal
setupterm	Sets up the TERMINAL structure for use by curses
endwin	Terminates the curses subroutine libraries and their data structures
isendwin	Returns TRUE if the **endwin** subroutine has been called without any subsequent calls to the **wrefresh** subroutine

You must include the **curses.h** file at the beginning of any program that calls curses subroutines. To do this, use the following statement:

```
#include <curses.h>
```

Before you can call subroutines that manipulate windows or screens, you must call the **initscr** or **newterm** subroutine. These subroutines first save the terminal's settings. These subroutines then call the **setupterm** subroutine to establish a curses terminal.

If you need to temporarily suspend curses, use a shell escape or system call for example. To resume after a temporary escape, you should call the **wrefresh** or **doupdate** subroutine. Before exiting a curses program, you must call the **endwin** subroutine. The **endwin** subroutine restores tty modes, moves the cursor to the lower left corner of the screen, and resets the terminal into the proper nonvisual mode.

Most interactive, screen-oriented programs require character-at-a-time input without echoing the result to the screen. To establish your program with character-at-a-time input, call the **cbreak** and **noecho** subroutines after calling the **initscr** subroutine. When accepting this type of input, programs should also call the following subroutines:

- **nonl** subroutine.
- **intrflush** subroutine with the *Window* parameter set to the **stdscr** and the *Flag* parameter set to **FALSE**. The *Window* parameter is required but ignored. You can use **stdscr** as the value of the *Window* parameter, because **stdscr** is already created for you.
- **keypad** subroutine with the *Window* parameter set to the **stdscr** and the *Flag* parameter set to **TRUE**.

The **isendwin** subroutine is helpful if, for optimization reasons, you don't want to call the **wrefresh** subroutine needlessly. You can determine if the **endwin** subroutine was called without any subsequent calls to the **wrefresh** subroutine by using the **isendwin** subroutine.

Windows in the Curses Environment

A curses program manipulates windows that appear on a terminal's display. A window can be as large as the entire display or as small as a single character in length and height.

> **Note:** Pads are the exception. A pad is a window that is not restricted by the size of the screen. For more information, see "Pads" on page 6.

Within a curses program, windows are variables declared as type WINDOW. The WINDOW data type is defined in the **/usr/include/curses.h** file as a C data structure. You create a window by allocating a portion of a machine's memory for a window structure. This structure describes the characteristics of the window. When a program changes the window data internally in memory, it must use the **wrefresh** subroutine (or equivalent subroutine) to update the external, physical screen to reflect the internal change in the appropriate window structure.

The Default Window Structure

Curses provides a virtual default window called stdscr. The stdscr represents, in memory, the entire terminal display. The stdscr window structure is created automatically when the curses library is initialized and it describes the display. When the library is initialized, the *length* and *width* variables are set to the length and width of the physical display.

Programs that use the stdscr first manipulate the stdscr and then call the **refresh** subroutine to refresh the external display so that it matches the stdscr window.

In addition to the stdscr, you can define your own windows. These windows are known as user-defined windows to distinguish them from the stdscr. Like the stdscr, user-defined windows exist in machine memory as structures. Except for the amount of memory available to a program, there is no limit to the number of windows you can create. A curses program can manipulate the default window, user-defined windows, or both.

The Current Window Structure

Curses supports another virtual window called curscr (current screen). The curscr window is an internal representation of what currently appears on the terminal's external display.

When a program requires the external representation to match the internal representation, it must call a subroutine, such as the **wrefresh** subroutine, to update the physical display (or the **refresh** subroutine if the program is working with the stdscr).

refresh, or **wrefresh**	Updates the terminal and curscr to reflect changes made to a window.

The curscr is reserved for internal use by curses. You should not manipulate the curscr.

Subwindows

Curses also allows you to construct subwindows. Subwindows are rectangular portions within other windows. A subwindow is also of type WINDOW. The window that contains a subwindow is known as the subwindow's parent and the subwindow is known as the containing window's child.

Changes to either the parent window or the child window within the area overlapped by the subwindow are made to both windows. After modifying a subwindow, you should call the **touchline** or **touchwin** subroutine on the parent window before refreshing it.

touchline	Forces a range of lines to be refreshed at the next call to the **wrefresh** subroutine.
touchwin	Forces every character in a window's character array to be refreshed at the next call of the **wrefresh** subroutine. The **touchwin** subroutine does not save optimization information. This subroutine is useful with overlapping windows.

A refresh called on the parent refreshes the children as well.

A subwindow can also be a parent window. The process of layering windows inside of windows is called nesting.

Before you can delete a parent window, you must first delete all of its children using the **delwin** subroutine.

delwin	Removes a window data structure.

Curses returns an error if you try to delete a window before removing all of its children.

Pads

A pad is a type of window that is not restricted by the terminal's display size or associated with a particular part of the display. Because a pad is usually larger than the physical display, only a portion of a pad is visible to the user at a given time.

Use pads if you have a large amount of related data that you want to keep all together in one window but you do not need to display all of the data at once.

Windows within pads are known as subpads. Subpads are positioned within a pad at coordinates relative to the parent pad. This placement differs from subwindows which are positioned using screen coordinates.

prefresh or **pnoutrefresh**	Updates the terminal and curscr to reflect changes made to a pad.

Unlike other windows, scrolling or echoing of input does not automatically refresh a pad. Like subwindows, when changing the image of a subpad, you must call either the **touchline** or **touchwin** subroutine on the parent pad before refreshing the parent.

You can use all the curses subroutines with pads except for the **newwin**, **subwin**, **wrefresh**, and **wnoutrefresh** subroutines. These subroutines are replaced with the **newpad**, **subpad**, **prefresh**, and **pnoutrefresh** subroutines.

Manipulating Window Data with Curses

When curses is initialized, the stdscr is provided automatically. You can manipulate the stdscr using the curses subroutine library or you can create your own, user-defined windows. This section discusses the following topics as they relate to manipulating window data:

Creating Windows

You can create your own window using the **newwin** subroutine.

newwin Creates a new window data structure.

Each time you call the **newwin** subroutine, curses allocates a new window structure in memory. This structure contains all the information associated with the new window. Curses does not put a limit on the number of windows you can create. The number of nested subwindows is limited to the amount of memory available up to the value of SHRT_MAX as defined in the **/usr/include/limits.h** file.

You can change windows without regard to the order in which they were created. Updates to the terminal's display occur through calls to the **wrefresh** subroutine.

Subwindows

subwin Creates a subwindow of an existing window.

You must supply coordinates for the subwindow relative to the terminal's display. The subwindow must fit within the bounds of the parent window; otherwise, a null value is returned.

Pads

newpad Creates a new pad data structure.
subpad Creates and returns a pointer to a subpad within a pad.

The new subpad is positioned relative to its parent.

Removing Windows, Pads, and Subwindows

To remove a window, pad, or subwindow, use the **delwin** subroutine. Before you can delete a window or pad, you must have already deleted its children; otherwise, the **delwin** subroutine returns an error.

Changing the Screen or Window Images

When curses subroutines change the appearance of a window, the internal representation of the window is updated while the display remains unchanged until the next call to the **wrefresh** subroutine. The **wrefresh** subroutine uses the information in the window structure to update the display.

Refreshing Windows

Any time you write output to a window or pad structure, you must refresh the terminal's display to match the internal representation. A refresh does the following:

- Compares the contents of the curscr to the contents of the user-defined or stdscr
- Updates the curscr structure to match the user-defined or stdscr
- Redraws the portion of the physical display that changed

refresh, or **wrefresh**	Updates the terminal and curscr to reflect changes made to a window.
wnoutrefresh or **doupdate**	Updates the designated windows and outputs them all at once to the terminal. These subroutines are useful for faster response when there are multiple updates.

The **refresh** and **wrefresh** subroutines first call the **wnoutrefresh** subroutine to copy the window being refreshed to the current screen. They then call the **doupdate** subroutine to update the display.

If you need to refresh multiple windows at the same time, use one of the two available methods. You can use a series of calls to the **wrefresh** subroutine that result in alternating calls to the **wnoutrefresh** and **doupdate** subroutines. You can also call the **wnoutrefresh** subroutine once for each window and then call the **doupdate** subroutine once. With the second method, only one burst of output is sent to the display.

Subroutines Used for Refreshing Pads

The **prefresh** and **pnoutrefresh** subroutines are similar to the **wrefresh** and **wnoutrefresh** subroutines.

prefresh or **pnoutrefresh**	Updates the terminal and curscr to reflect changes made to a user-defined pad.

The **prefresh** subroutine updates both the current screen and the physical display, while the **pnoutrefresh** subroutine updates curscr to reflect changes made to a user-defined pad. Because pads instead of windows are involved, these subroutines require additional parameters to indicate which part of the pad and screen are involved.

Refreshing Areas that Have Not Changed

During a refresh, only those areas that have changed are redrawn on the display. It is possible to refresh areas of the display that have not changed using the **touchwin** and **touchline** subroutines.

touchline	Forces a range of lines to be refreshed at the next call to the **wrefresh** subroutine.
touchwin	Forces every character in a window's character array to be refreshed at the next call of the **wrefresh** subroutine. The **touchwin** subroutine does not save optimization information. This subroutine is useful with overlapping windows.

Combining the **touchwin** and **wrefresh** subroutines is helpful when dealing with subwindows or overlapping windows. To bring a window forward from behind another window, call the **touchwin** subroutine followed by the **wrefresh** subroutine.

Garbled Displays

If text is sent to the terminal's display with a noncurses subroutine, such as the **echo** or **printf** subroutine, the external window can become garbled. In this case, the display changes, but the current screen is not updated to reflect these changes. Problems can arise when a refresh is called on the garbled screen because after a screen is garbled, there is no difference between the window being refreshed and the current screen structure. As a result, spaces on the display caused by garbled text are not changed.

A similar problem can also occur when a window is moved. The characters sent to the display with the noncurses subroutines do not move with the window internally.

If the screen becomeS garbled, call the **wrefresh** subroutine on the curscr to update the display to reflect the current physical display.

Manipulating Window Content

After a window or subwindow is created, programs often must manipulate them in some way.

box	Draws a box in or around a window.
copywin	Provides more precise control over the **overlay** and **overwrite** subroutine.
garbagedlines	Indicates to curses that a screen line is garbaged and should be thrown away before having anything written over the top of it.
mvwin	Moves a window or subwindow to a new location.
overlay or **overwrite**	Copies one window on top of another.
ripoffline	Removes a line from the default screen.

The **mvwin** subroutine moves a window or subwindow. The **box** subroutine draws a box around the edge of a window or subwindow.

To use the **overlay** and **overwrite** subroutines, the two windows must overlap. Also, be aware that the **overwrite** subroutine is destructive whereas the **overlay** subroutine is not. When text is copied from one window to another using the **overwrite** subroutine, blank portions from the copied window overwrite any portions of the window copied to. The **overlay** subroutine is nondestructive because it does not copy blank portions from the copied window.

Similar to the **overlay** and **overwrite** subroutines, the **copywin** subroutine allows you to copy a portion of one window to another. Unlike **overlay** and **overwrite** subroutines, the windows do not have to overlap for you to use the **copywin** subroutine.

You can use the **ripoffline** subroutine to remove a line from the stdscr. If you pass this subroutine a positive *line* argument, the specified number of lines is removed from the top of the stdscr. Otherwise, if you pass the subroutine a negative *line* argument, the lines are removed from the bottom of the stdscr.

Finally, you can use the **garbagedlines** subroutine to discard a specified range of lines before writing anything new.

Support for Filters

The **filter** subroutine is provided for curses applications that are filters.

filter	Sets the size of the terminal screen to 1 line.

This subroutine causes curses to operate as if the stdscr was only a single line on the screen. When running with the **filter** subroutine, curses does not use any terminal capabilities that require knowledge of the line that curses is on.

Controlling the Cursor with Curses

In the Curses library, there are two types of cursors:

logical cursor	The cursor location within each window. A window's data structure keeps track of the location of its logical cursor. Each window has a logical cursor.
physical cursor	The display cursor. The workstation uses this cursor to write to the display. There is only one physical cursor per display.

You can only add to or erase characters at the current cursor location in a window. The following subroutines are provided for controlling the cursor:

move	Moves the logical cursor associated with the stdscr.
wmove	Moves the logical cursor associated with a user-defined window.
getbegyx	Places the beginning coordinates of the window in integer variables *y* and *x*.
getmaxyx	Places the size of the window in integer variables *y* and *x*.
getsyx	Returns the current coordinates of the virtual screen cursor.
getyx	Returns the position of the logical cursor associated with a specified window.
leaveok	Controls physical cursor placement after a call to the **wrefresh** subroutine.
mvcur	Moves the physical cursor.
setsyx	Sets the virtual screen cursor to the specified coordinate.

After a call to the **refresh** or **wrefresh** subroutine, curses places the physical cursor at the last updated character position in the window. To leave the physical cursor where it is and not move it after a refresh, call the **leaveok** subroutine with the *Window* parameter set to the desired window and the *Flag* parameter set to **TRUE**.

Manipulating Characters with Curses

You can add characters to a curses window using a keyboard or a curses application. This section provides an overview of the ways you can add, remove, or change characters that appear in a curses window.

Character Size

Historically, a position on the screen has corresponded to a single stored byte. This correspondence is no longer true for several reasons:

- Some characters may occupy several columns when displayed on the screen.
- Some characters may be non-spacing characters, defined only in association with a spacing character.
- The number of bytes to hold a character from the extended character sets depends on the LC_CTYPE locale category.

Some character sets define multi-column characters that occupy more than one column position when displayed on the screen.

Writing a character whose width is greater than the width of the destination window is an error.

Adding Characters to the Screen Image

The Curses library provides a number of subroutines that write text changes to a window and mark the area to be updated at the next call to the **wrefresh** subroutine.

waddch Subroutines

The **waddch** subroutines overwrite the character at the current logical cursor location with a specified character. After overwriting, the logical cursor is moved one space to the right. If the **waddch** subroutines are called at the right margin, these subroutines also add an automatic newline character. Additionally, if you call one of these subroutines at the bottom of a scrolling region and scrollok is enabled, the region is scrolled up one line. For example, if you added a new line at the bottom line of a window, the window would scroll up one line.

If the character to add is a tab, newline, or backspace character, curses moves the cursor appropriately in the window to reflect the addition. Tabs are set at every eighth column. If the character is a newline, curses first uses the **wclrtoeol** subroutine to erase the current line from the logical cursor position to the end of the line before moving the cursor. The **waddch** subroutine family is made up of the following:

waddch subroutine	Adds a character to the user-defined window.

addch macro	Adds a character to the stdscr.
mvaddch macro	Moves a character to the specified location before adding it to the stdscr.
mvwaddch macro	Moves a character to the specified location before adding it to the user-defined window.

By using the **winch** and **waddch** subroutine families together, you can copy text and video attributes from one place to another. Using the **winch** subroutine family, you can retrieve a character and its video attributes. You can then use one of the **waddch** subroutines to add the character and its attributes to another location.

You can also use the **waddch** subroutines to add control characters to a window. Control characters are drawn in the ^X notation.

> **Note:** Calling the **winch** subroutine on a position in the window containing a control character does not return the character. Instead, it returns one character of the control character representation.

Outputting Single, Noncontrol Characters: When outputting single, noncontrol characters, there is significant performance gain to using the **wechochar** subroutines. These subroutines are functionally equivalent to a call to the corresponding **waddchr** subroutine followed by the corresponding **wrefresh** subroutine. The **wechochar** subroutines include the **wechochar** subroutine, the **echochar** macro, and the **pechochar** subroutine.

Some character sets may contain non-spacing characters. (Non-spacing characters are those, other than the ' \ 0 ' character, for which **wcwidth ( )** returns a width of zero.) The application may write non-spacing characters to a window. Every non-spacing character in a window is associated with a spacing character and modifies the spacing character. Non-spacing characters in a window cannot be addressed separately. A non-spacing character is implicitly addressed whenever a Curses operation affects the spacing character with which the non-spacing character is associated.

Non-spacing characters do not support attributes. For interfaces that use wide characters and attributes, the attributes are ignored if the wide character is a non-spacing character. Multi-column characters have a single set of attributes for all columns. The association of non-spacing characters with spacing characters can be controlled by the application using the wide character interfaces. The wide character string functions provide codeset-dependent association.

Two typical effects of a non-spacing character associated with a spacing character called c, are as follows:

- The non-spacing character may modify the appearance of c. (For instance, there may be non-spacing characters that add diacritical marks to characters. However, there may also be spacing characters with built-in diacritical marks.)
- The non-spacing characters may bridge c to the character following c. (Examples of this usage are the formation of ligatures and the conversion of characters into compound display forms, words, or ideograms.)

Implementations may limit the number of non-spacing characters that can be associated with a spacing character, provided any limit is at least 5.

Complex Characters

A complex character is a set of associated characters, which may include a spacing character and may also include any non-spacing characters associated with it. A spacing complex character is a complex character that includes one spacing character and any non-spacing characters associated with it. An example of a code set that has complex characters is ISO/IEC 10646-1:1993.

A complex character can be written to the screen. If the complex character does not include a spacing character, any non-spacing characters are associated with the spacing complex character that exists at the specified screen position. When the application reads information back from the screen, it obtains spacing complex characters.

The cchar_t data type represents a complex character and its rendition. When a cchar_t represents a non-spacing complex character (that is, when there is no spacing character within the complex character), then its rendition is not used; when it is written to the screen, it uses the rendition specified by the spacing character already displayed.

An object of type cchar_t can be initialized using setchar( ) and its contents can be extracted using getchar( ). The behavior of functions that take a cchar_t value that was not initialized in this way or obtained from a curses function that has a cchar_t output argument.

Special Characters

Some functions process special characters as specified below.

In functions that do not move the cursor based on the information placed in the window, these special characters would only be used within a string in order to affect the placement of subsequent characters; the cursor movement specified below does not persist in the visible cursor beyond the end of the operation. In functions that do not move the cursor, these special characters can be used to affect the placement of subsequent characters and to achieve movement of the physical cursor.

Backspace	Unless the cursor was already in column 0, Backspace moves the cursor one cloumn toward the start of the current line and any characters after the Backspace are added or inserted starting there.
Carriage return	Unless the cursor was already in column 0, Carriage return moves the cursor to the start of the current line. Any characters after the Carriage return are added or inserted starting there.
newline	In an add operation, curses adds the background character into successive columns until reaching the end of the line. Scrolling occurs, and any characters after the newline character are added, starting at the beginning of the new line. In an insert operation, newline erases the remainder of the current line with the background character, effectively a wclrtoeol ( ), and moves the cursor to the start of a new line. When scrolling is enabled, advancing the cursor to a new line may cause scrolling. Any characters after the newline character are inserted at the beginning of the new line. The filter ( ) function may inhibit this processing.
Tab	Tab characters in text move subsequent characters to the next horizontal tab stop. By default, tab stops are in columns 0, 8, 16, and so on. In an insert or add operation, curses inserts or adds, respectively, the background character into successive columns until reaching the next tab stop. If there are no more tab stops in the current line, wrapping and scrolling occur.

Control Characters: The curses functions that perform special-character processing conceptually convert control characters to the (' ^ ') character followed by a second character (which is an upper-case letter if it is alphabetic) and write this string to the window in place of the control character. The funtions that retrieve text from the window will not retrieve the original control character.

Line Graphics: You can use the following variables to add line-drawing characters to the screen with the **waddch** subroutine. When defined for the terminal, the variable will have the **A_ALTCHARSET** bit turned on. Otherwise, the default character listed in the following table will be stored in the variable.

Variable Name	Default Character	Glyph Description

ACS_ULCORNER	+	upper left corner
ACS_LLCORNER	+	lower left corner
ACS_URCORNER	+	upper right corner
ACS_LRCORNER	+	lower right corner
ACS_RTEE	+	right tee
ACS_LTEE	+	left tee
ACS_BTEE	+	bottom tee
ACS_TTEE	+	top tee
ACS_HLINE	—	horizontal line
ACS_VLINE	|	vertical line
ACS_PLUS	+	plus
ACS_S1	-	scan line 1
ACS_S9	_	scan line 9
ACS_DIAMOND	+	diamond
ACS_CKBOARD	:	checker board (stipple)
ACS_DEGREE	,	degree symbol
ACS_PLMINUS	#	plus/minus
ACS_BULLET	o	bullet
ACS_LARROW	<	arrow pointing left
ACS_RARROW	>	arrow pointing right
ACS_DARROW	v	arrow pointing down
ACS_UARROW	^	arrow pointing up
ACS_BOARD	#	board of squares
ACS_LANTERN	#	lantern symbol
ACS_BLOCK	#	solid square block

waddstr Subroutines

The **waddstr** subroutines add a null-terminated character string to a window, starting with the current character. If you are adding a single character, use the **waddch** subroutine. Otherwise, use the **waddstr** subroutine. The following are part of the **waddstr** subroutine family:

waddstr subroutine	Adds a character string to a user-defined window.
addstr macro	Adds a character string to the stdscr.
mvaddstr macro	Moves the logical cursor to a specified location before adding a character string to the stdscr.
wmvaddstr macro	Moves the logical cursor to a specified location before adding a character string to a user-defined window.

winsch Subroutines

The **winsch** subroutines insert a specified character before the current character in a window. All characters to the right of the inserted character are moved one space to the right. As a result, the rightmost character on the line may be lost. The positions of the logical and physical cursors do not change after the move. The **winsch** subroutines include the following:

winsch subroutine	Inserts a character in a user-defined window.

insch macro	Inserts a character in the stdscr.
mvinsch macro	Moves the logical cursor to a specified location in the stdscr before inserting a character.
mvwinsch macro	Moves the logical cursor to a specified location in a user-defined window before inserting a character.

winsertln Subroutines

The **winsertln** subroutines insert a blank line above the current line in a window. The **insertln** subroutine inserts a line in the stdscr. The bottom line of the window is lost. The **winsertln** subroutine performs the same action in a user-defined window.

wprintw Subroutines

The **wprintw** subroutines replace a series of characters (starting with the current character) with formatted output. The format is the same as for the **printf** command. The following subroutine and macros belong to the **printw** family:

wprintw subroutine	Replaces a series of characters in a user-defined window.
printw macro	Replaces a series of characters in the stdscr.
mvprintw macro	Moves the logical cursor to a specified location in the stdscr before replacing any characters.
mvwprintw macro	Moves the logical cursor to a specified location in a user-defined window before replacing any characters.

The **wprintw** subroutines make calls to the **waddch** subroutine to replace characters.

unctrl Macro

The **unctrl** macro returns a printable representation of the specified control character, displayed in the ^X notation. The **unctrl** macro returns print characters as is.

Enabling Text Scrolling

idlok	Allows curses to use the hardware insert/delete line feature.
scrollok	Enables a window to scroll when the cursor is moved off the right edge of the last line of a window.
setscrreg or **wsetscrreg**	Sets a software scrolling region within a window.

Scrolling occurs when a program or user moves a cursor off a window's bottom edge. For scrolling to occur, you must first use the **scrollok** subroutine to enable scrolling for a window. A window is scrolled if scrolling is enabled and if any of the following occurs:

- The cursor is moved off the edge of a window.
- A new-line character is encountered on the last line.
- A character is inserted in the last position of the last line.

When a window is scrolled, curses will update both the window and the display. However, to get the physical scrolling effect on the terminal, you must call the **idlok** subroutine with the *Flag* parameter set to **TRUE**.

If scrolling is disabled, the cursor is left on the bottom line at the location where the character was entered.

When scrolling is enabled for a window, you can use the **setscrreg** subroutines to create a software scrolling region inside the window. You pass the **setscrreg** subroutines values for the top line and bottom

line of the region. If setscrreg is enabled for the region and scrolling is enabled for the window, any attempt to move off the specified bottom line causes all the lines in the region to scroll up one line. You can use the **setscrreg** macro to define a scrolling region in the stdscr. Otherwise, you use the **wsetscrreg** subroutine to define scrolling regions in user-defined windows.

> **Note:** Unlike the **idlok** subroutine, the **setscrreg** subroutines have nothing to do with the use of the physical scrolling region capability that the terminal may or may not have.

Deleting Characters

You can delete text by replacing it with blank spaces or by removing characters from a character array and sliding the rest of the characters on the line one space to the left.

werase Subroutines

erase or **werase**	Copies blank spaces to every position in a window.

The **erase** macro copies blank space to every position in the stdscr. The **werase** subroutine puts a blank space at every position in a user-defined window. To delete a single character in a window, use the **wdelch** subroutine.

wclear Subroutines

clear, or **wclear**	Clears the screen and sets a clear flag for the next refresh.
clearok	Determines whether curses clears a window on the next call to the **refresh** or **wrefresh** subroutine.

The **wclear** subroutines are similar to the **werase** subroutines. However, in addition to putting a blank space at every position of a window, the **wclear** subroutines also call the **wclearok** subroutine. As a result, the screen is cleared on the next call to the **wrefresh** subroutine.

The **wclear** subroutine family contains the **wclear** subroutine, the **clear** macro, and the **clearok** subroutine. The **clear** macro puts a blank at every position in the stdscr.

wclrtoeol Subroutines

clrtoeol or **wclrtoeol**	Erases the current line to the right of the logical cursor.

The **clrtoeol** macro operates in the stdscr, while the **wclrtoeol** subroutine performs the same action within a user-defined window.

wclrtobot Subroutines

clrtobot or **wclrtobot**	Erases the lines below and to the right of the logical cursor.

The **clrtobot** macro operates in the stdscr, while the **wclrtobot** performs the same action in a user-defined window.

wdelch Subroutines

wdelch subroutine	Deletes the current character in a user-defined window.
delch macro	Deletes the current character from the stdscr.
mvdelch macro	Moves the logical cursor before deleting a character from the stdscr.
mvwdelch macro	Moves the logical cursor before deleting a character from a user-defined window.

The **wdelch** subroutines delete the current character and move all the characters to the right of the current character on the current line one position to the left. The last character in the line is filled with a blank. The **delch** subroutine family consists of the following subroutine and macros:

wdeleteln Subroutines

deleteln or **wdeleteln**	Deletes the current line.

The **wdeleteln** subroutines delete the current line and move all lines below the current line up one line. This clears the window's bottom line.

Getting Characters

Your program can retrieve characters from the keyboard or from the display. The **wgetch** subroutines retrieve characters from the keyboard. The **winch** subroutines retrieve characters from the display.

wgetch Subroutines

The **wgetch** subroutines read characters from the keyboard attached to the terminal associated with the window. Before getting a character, these subroutines call the **wrefresh** subroutines if anything in the window has changed: for example, if the cursor has moved or text has changed. If the **wgetch** subroutine encounters a Ctrl-D key sequence during processing, it returns.

The following belong to the **wgetch** subroutine family:

wgetch subroutine	Gets a character from a user-defined window.
getch macro	Gets a character from the stdscr.
mvgetch macro	Moves the cursor before getting a character from the stdscr.
mvwgetch macro	Moves the cursor before getting a character from a user-defined window.

To place a character previously obtained by a call to the **wgetch** subroutine back in the input queue, use the **ungetch** subroutine. The character is retrieved by the next call to the **wgetch** subroutine.

The Importance of Terminal Modes: The output of the **wgetch** subroutines is, in part, determined by the mode of the terminal. The following list describes the action of the **wgetch** subroutines in each type of terminal mode:

Mode	Action of wgetch Subroutines
NODELAY	Returns a value of ERR if there is no input waiting.
DELAY	Stops reading until the system passes text through the program. If CBREAK mode is also set, the program stops after one character. If CBREAK mode is not set (NOCBREAK mode), the **wgetch** subroutine stops reading after the first new-line character. If ECHO is set, the character is also echoed to the window.
HALF-DELAY	Stops reading until a character is typed or a specified timeout is reached. If ECHO mode is set, the character is also echoed to the window.

> **Note:** When you use the **wgetch** subroutines do not set both the NOCBREAK mode and the ECHO mode at the same time. Setting both modes can cause undesirable results depending on the state of the tty driver when each character is typed.

Function Keys: Function keys are defined in the **curses.h** file. Function keys can be returned by the **wgetch** subroutine if the keypad is enabled. A terminal may not support all of the function keys. To see if a terminal supports a particular key, check its **terminfo** database definition. The following table lists the function keys defined in the **curses.h** file:

Name	Key Name

KEY_BREAK	Break key (unreliable).
KEY_DOWN	Down arrow key.
KEY_UP	Up arrow key.
KEY_LEFT	Left arrow key.
KEY_RIGHT	Right arrow key.
KEY_HOME	Home key (upward + left arrow).
KEY_BACKSPACE	Backspace (unreliable).
KEY F0	Function keys. Space for 64 keys is reserved.
KEYF(n)	Formula for $f^{n.}$
KEY_DL	Delete line.
KEY_IL	Insert line.
KEY_DC	Delete character.
KEY_IC	Insert character or enter insert mode.
KEY_EIC	Exit insert character mode.
KEY_CLEAR	Clear screen.
KEY_EOS	Clear to end of screen.
KEY_EOL	Clear to end of line.
KEY_SF	Scroll 1 line forward.
KEY_SR	Scroll 1 line backwards (reverse).
KEY_NPAGE	Next page.
KEY_PPAGE	Previous page.
KEY_STAB	Set tab.
KEY_CTAB	Clear tab.
KEY_CATAB	Clear all tabs.
KEY_ENTER	Enter or send.
KEY_SRESET	Soft (partial) reset.
KEY_RESET	Reset or hard reset.
KEY_PRINT	Print or copy.
KEY_IL	Home down or bottom (lower left) keypad.
KEY_A1	Upper left of keypad.
KEY_A3	Upper right of keypad.
KEY_B2	Center of keypad.
KEY_C1	Lower left of keypad.
KEY_C3	Lower right of keypad.
KEY_BTAB	Back tab key.
KEY_BEG	Beginning key.
KEY_CANCEL	Cancel key.
KEY-CLOSE	Close key.
KEY_COMMAND	Command key.
KEY_COPY	Copy key.
KEY_CREATE	Create key.
KEY_END	End key.

KEY_EXIT	Exit key.
KEY_FIND	Find key.
KEY_HELP	Help key.
KEY_MARK	Mark key.
KEY_MESSAGE	Message key.
KEY_MOVE	Move key.
KEY_NEXT	Next object key.
KEY_OPEN	Open key.
KEY_OPTIONS	Options key.
KEY_PREVIOUS	Previous object key.
KEY_REDO	Redo key.
KEY_REFERENCE	Reference key.
KEY_REFRESH	Refresh key.
KEY_REPLACE	Replace key.
KEY_RESTART	Restart key.
KEY_RESUME	Resume key.
KEY_SAVE	Save key.
KEY_SBEG	Shifted beginning key.
KEY_SCANCEL	Shifted cancel key.
KEY_SCOMMAND	Shifted command key.
KEY_SCOPY	Shifted copy key.
KEY_SCREATE	Shifted create key.
KEY_SDC	Shifted delete-character key.
KEY_SDL	Shifted delete-line key.
KEY_SELECT	Select key.
KEY_SEND	Shifted end key.
KEY_SEOL	Shifted clear-line key.
KEY_SEXIT	Shifted exit key.
KEY_SFIND	Shifted find key.
KEY_SHELP	Shifted help key.
KEY_SHOME	Shifted home key.
KEY_SIC	Shifted input key.
KEY_SLEFT	Shifted left arrow key.
KEY_SMESSAGE	Shifted message key.
KEY_SMOVE	Shifted move key.
KEY_SNEXT	Shifted next key.
KEY_SOPTIONS	Shifted options key.
KEY_SPREVIOUS	Shifted previous key.
KEY_SPRINT	Shifted print key.
KEY_SREDO	Shifted redo key.
KEY_SREPLACE	Shifted replace key.
KEY_SRIGHT	Shifted right arrow key.

KEY_SRSUME	Shifted resume key.
KEY_SSAVE	Shifted save key.
KEY_SSUSPEND	Shifted suspend key.
KEY_SUNDO	Shifted undo key.
KEY_SUSPEND	Suspend key.
KEY_UNDO	Undo key.

Getting Function Keys: If your program enables the keyboard with the **keypad** subroutine, and the user presses a function key, the token for that function key is returned instead of raw characters. The possible function keys are defined in the **/usr/include/curses.h** file. Each define statement begins with a **KEY_** prefix and the keys are defined as integers beginning with the value 03510.

If a character is received that could be the beginning of a function key (such as an Escape character), curses sets a timer (a structure of type timeval that is defined in **/usr/include/sys/time.h**). If the remainder of the sequence is not received before the timer expires, the character is passed through. Otherwise, the function key's value is returned. For this reason, after a user presses the escape key there is a delay before the escape is returned to the program. You should avoid using the escape key where possible when you call a single-character subroutine such as the **wgetch** subroutine. This timer can be overridden or extended by the use of the environment variable **ESCDELAY**.

The **ESCDELAY** environment variable sets the length of time to wait before timing out and treating the ESC keystroke as the Escape character rather than combining it with other characters in the buffer to create a key sequence. The **ESCDELAY** value is measured in fifths of a millisecond. If the **ESCDELAY** variable is 0, the system immediately composes the Escape response without waiting for more information from the buffer. You may choose any value from 0 to 99,999. The default setting for the ESCDELAY variable is 500 (1/10th of a second).

To prevent the **wgetch** subroutine from setting a timer, call the **notimeout** subroutine. If notimeout is set to TRUE, curses does not distinguish between function keys and characters when retrieving data.

keyname Subroutine

The **keyname** subroutine returns a pointer to a character string containing a symbolic name for the *Key* argument. The *Key* argument can be any key returned from the **wgetch**, **getch**, **mvgetch**, or **mvwgetch** subroutines.

winch Subroutines

The **winch** subroutines retrieve the character at the current position. If any attributes are set for the position, the attribute values are ORed into the value returned. You can use the **winch** subroutines to extract only the character or its attributes. To do this, use the predefined constants **A_CHARTEXT** and **A_ATTRIBUTES** with the logical & (ampersand) operator. These constants are defined in the **curses.h** file. The following are the **inch** subroutines:

winch subroutine	Gets the current character from a user-defined window.
inch macro	Gets the current character from the stdscr.
mvinch macro	Moves the logical cursor before calling the **inch** subroutine on the stdscr.
mvwinch macro	Moves the logical cursor before calling the **winch** subroutine in the user-defined window.

wscanw Subroutines

The **wscanw** subroutines read character data, interpret it according to a conversion specification, and store the converted results into memory. The **wscanw** subroutines use the **wgetstr** subroutines to read the character data. The following are the **wscanw** subroutines:

wscanw subroutine	Scans a user-defined window.
scanw macro	Scans the stdscr.
mvscanw macro	Moves the logical cursor before scanning the stdscr.
mvwscanw macro	Moves the logical cursor in the user-defined window before scanning.

The **vwscanw** subroutine scans a window using a variable argument list. For information about manipulating variable argument lists, see the **varargs** macros.

Understanding Terminals with curses

The capabilities of your program are limited, in part, by the capabilities of the terminal on which it runs. This section provides information about initializing terminals and identifying their capabilities.

Manipulating Multiple Terminals

With curses, you can use one or more terminals for input and output. The terminal subroutines enable you to establish new terminals, to switch input and output processing, and to retrieve terminal capabilities.

You can start curses on a single default screen using the **initscr** subroutine. This should be sufficient for most applications. However, if your application sends output to more than one terminal, you should use the **newterm** subroutine. Call this subroutine for each terminal. You should also use the **newterm** subroutine if your application wants an indication of error conditions so that it can continue to run in a line-oriented mode if the terminal cannot support a screen-oriented program.

When it completes, a program must call the **endwin** subroutine for each terminal it used. If you call the **newterm** subroutine more than once for the same terminal, the first terminal referred to must be the last one for which you call the **endwin** subroutine.

The **set_term** subroutine switches input and output processing between different terminals.

Determining Terminal Capabilities

curses supplies the following subroutines to help you determine the capabilities of a terminal:

longname	Returns the verbose name of the terminal.
has_ic	Determines whether a terminal has the insert-character capability.
has_il	Determines whether a terminal has the insert-line capability.

The **longname** subroutine returns a pointer to a static area containing a verbose description of the current terminal. This area is defined only after a call to the **initscr** or **newterm** subroutine. If you intend to use the **longname** subroutine with multiple terminals, you should know that each call to the **newterm** subroutine overwrites this area. Calls to the **set_term** subroutine do not restore the value. Instead, save this area between calls to the **newterm** subroutine.

The **has_ic** subroutine returns TRUE if the terminal has insert and delete character capabilities.

The **has_il** subroutine returns TRUE if the terminal has insert and delete line capabilities or can simulate the capabilities using scrolling regions. Use the **has_il** subroutine to check whether it is appropriate to turn on physical scrolling using the **scrollok** or **idlok** subroutines.

Setting Terminal Input and Output Modes

The subroutines that control input and output determine how your application retrieves and displays data to users.

Input Modes

Special input characters include the flow-control characters, the interrupt character, the erase character, and the kill character. Four mutually-exclusive curses modes let the application control the effect of the input characters.

Cooked Mode

This achieves normal line-at-a-time processing with all special characters handled outside the application. This achieves the same effect as canonical-mode input processing. The state of the ISIG and IXON flags are not changed upon entering this mode by calling nocbreak( ), and are set upon entering this mode by calling noraw( ).

The implementation supports erase and kill characters from any supported locale, no matter what the width of the character is.

cbreak Mode

Characters typed by the user are immediately available to the application and curses does not perform special processing on either the erase character or the kill character. An application can select cbreak mode to do its own line editing but to let the abort character be used to abort the task. This mode achieves the same effect as non-canonical-mode, Case B input processing (with MIN set to 1 and ICRNL cleared). The state of the ISIG and IXON flags are not changed upon entering this mode.

Half-Delay Mode

The effect is the same as **cbreak**, except that input functions wait until a character is available or an interval defined by the application elapses, whichever comes first. This mode achieves the same effect as non-canonical-mode, Case C input processing (with TIME set to the value specified by the application). The state of the ISIG and IXON flags are not changed upon entering this mode.

Raw Mode

Raw mode gives the application maximum control over terminal input. The application sees each character as it is typed. This achieves the same effect as non-canonical mode, Case D input processing. The ISIG and IXON flags are cleared upon entering this mode.

The terminal interface settings are recorded when the process calls **initscr** or **newterm** to initialize curses and restores these settings when **endwin** is called. The initial input mode for curses operations is unspecified unless the implementation supports Enhanced curses compliance, in which the initial input mode is **cbreak** mode.

The behavior of the BREAK key depends on other bits in the display driver that are not set by curses.

Delay Mode

Two mutually-exclusive delay modes specify how quickly certain curses functions return to the application when there is no terminal input waiting when the function is called:

No Delay	The function fails.
Delay	The application waits until the implementation passes text through to the application. If cbreak or Raw Mode is set, this is after one character. Otherwise, this is after the first <newline> character, end-of-line character, or end-of-file character.

The effect of No Delay Mode on function key processing is unspecified.

Echo Processing

echo mode determines whether curses echoes typed characters to the screen. The effect of echo mode is analogous to the effect of the echo flag in the local mode field of the termios structure associated with the terminal device connected to the window. However, curses always clears the echo flag when invoked, to inhibit the operating system from performing echoing. The method of echoing characters is not identical to the operating system's method of echoing characters, because curses performs additional processing of terminal input.

If in echo mode, curses performs its own echoing. Any visible input character is stored in the current or specified window by the input function that the application called, at that window's cursor position, as though addch( ) were called, with all consequent effects such as cursor movement and wrapping.

If not in echo mode, any echoing of input must be performed by the application. Applications often perform their own echoing in a controlled area of the screen, or do not echo at all, so they disable echo mode.

It may not be possible to turn off echo processing for synchronous and network asynchronous terminals because echo processing is done directly by the terminals. Applications running on such terminals should be aware that any characters typed will appear on the screen at wherever the cursor is positioned.

cbreak or **nocbreak**	Puts the terminal into or takes it out of CBREAK mode.
delay_output	Sets the output delay in milliseconds.
echo or **noecho**	Controls echoing of typed characters to the screen.
halfdelay	Returns ERR if no input was typed after blocking for a specified amount of time.
nl or **nonl**	Determines whether curses translates a new line into a carriage return and line feed on output, and translates a return into a new line on input.
raw or **noraw**	Places the terminal into or out of RAW mode.

The **echo** subroutine puts the terminal into echo mode. In echo mode, curses writes characters typed by the user to the terminal at the physical cursor position. The **noecho** subroutine takes the terminal out of echo mode.

The **raw** subroutine puts the terminal into raw mode. In raw mode, characters typed by the user are immediately available to the program. Additionally, the interrupt, quit, suspend, and flow-control characters are passed uninterpreted instead of generating a signal as they do in cbreak mode. The **noraw** subroutine takes the terminal out of raw mode.

The **cbreak** subroutine performs a subset of the functions performed by the **raw** subroutine. In cbreak mode, characters typed by the user are immediately available to the program and erase or kill character processing is not done. Unlike RQW mode, interrupt and flow characters are acted upon. Otherwise, the tty driver buffers the characters typed until a new line or carriage return is typed.

> **Note:** cbreak mode disables translation by the tty driver.

The **nocbreak** subroutine takes the terminal out of cbreak mode.

The **delay_output** subroutine sets the output delay to the specified number of milliseconds. Do not use this subroutine extensively because it uses padding characters instead of a processor pause.

The **nl** and **nonl** subroutines, respectively, control whether curses translates new lines into carriage returns and line feeds on output, and whether curses translates carriage returns into new lines on input. Initially, these translations do occur. By disabling these translations, the curses subroutine library has more control over the line-feed capability, resulting in faster cursor motion.

Using the terminfo and termcap Files

When curses is initialized, it checks the **TERM** environment variable to identify the terminal type. Then, curses looks for a definition explaining the capabilities of the terminal. Usually this information is kept in a local directory specified by the **TERMINFO** environment variable or in the **/usr/share/lib/terminfo** directory. All curses programs first check to see if the **TERMINFO** environment variable is defined. If this variable is not defined, the **/usr/share/lib/terminfo** directory is checked.

For example, if the **TERM** variable is set to `vt100` and the **TERMINFO** variable is set to the **/usr/mark/myterms** file, curses checks for the **/usr/mark/myterms/v/vt100** file. If this file does not exist, curses checks the **/usr/share/lib/terminfo/v/vt100** file.

Additionally, the **LINES** and **COLUMNS** environment variables can be set to override the terminal description.

Writing Programs That Use the terminfo Subroutines

Use the **terminfo** subroutines when your program needs to deal directly with the terminfo database. For example, use these subroutines to program function keys. In all other cases, curses subroutines are more suitable and their use is recommended.

Initializing Terminals: Your program should begin by calling the **setupterm** subroutine. Normally, this subroutine is called indirectly by a call to the **initscr** or **newterm** subroutine. The **setupterm** subroutine reads the terminal-dependent variables defined in the **terminfo** database. The **terminfo** database includes boolean, numeric, and string variables. All of these **terminfo** variables use the values defined for the specified terminal. After reading the database, the **setupterm** subroutine initializes the **cur_term** variable with the terminal definition. When working with multiple terminals, you can use the **set_curterm** subroutine to set the **cur_term** variable to a specific terminal.

Another subroutine, **restartterm**, is similar to the **setupterm** subroutine. However, it is called after memory is restored to a previous state. For example, you would call the **restartterm** subroutine after a call to the **scr_restore** subroutine. The **restartterm** subroutine assumes that the input and output options are the same as when memory was saved, but that the terminal type and baud rate may differ.

The **del_curterm** subroutine frees the space containing the capability information for a specified terminal.

Header Files: Include the **curses.h** and **term.h** files in your program in the following order:

```
#include <curses.h>
#include <term.h>
```

These files contain the definitions for the strings, numbers, and flags in the **terminfo** database.

Handling Terminal Capabilities: Pass all parametized strings through the **tparm** subroutine to instantiate them. You should print all **terminfo** strings and the output of the **tparm** subroutine with the **tputs** or **putp** subroutine.

putp	Provides a shortcut to the **tputs** subroutine.
tparm	Instantiates a string with parameters.
tputs	Applies padding information to the given string and outputs it.

Use the following subroutines to obtain and pass terminal capabilities:

tigetflag	Returns the value of a specified boolean capability. If the capability is not boolean, a -1 is returned.
tigetnum	Returns the value of a specified numeric capability. If the capability is not numeric, a -2 is returned.
tigetstr	Returns the value of a specified string capability. If the capability specified is not a string, the **tigetstr** subroutine returns the value of (**char ***) -1.

Exiting the Program: When your program exits you should restore the tty modes to their original state. To do this, call the **reset_shell_mode** subroutine. If your program uses cursor addressing, it should output the **enter_ca_mode** string at startup and the **exit_ca_mode** string when it exits.

Programs that use shell escapes should call the **reset_shell_mode** subroutine and output the **exit_ca_mode** string before calling the shell. After returning from the shell, the program should output the

enter_ca_mode string and call the **reset_prog_mode** subroutine. This process differs from standard curses operations which call the **endwin** subroutine on exit.

Low-Level Screen Subroutines

Use the following subroutines for low-level screen manipulations:

scr_restore	Restores the virtual screen to the contents of a previously dumped file.
scr_dump	Dumps the contents of the virtual screen to the specified file.
scr_init	Initializes the curses data structures from a specified file.
ripoffline	Strips a single line from the stdscr.

termcap Subroutines

If your program uses the **termcap** file for terminal information, the **termcap** subroutines are included as a conversion aid. The parameters are the same for the **termcap** subroutines. curses emulates the subroutines using the **terminfo** database. The following **termcap** subroutines are supplied:

tgetent	Emulates the **setupterm** subroutine.
tgetflag	Returns the boolean entry for a termcap identifier.
tgetnum	Returns the numeric entry for a termcap identifier.
tgetstr	Returns the string entry for a termcap identifier.
tgoto	Duplicates the **tparm** subroutine. The output from the **tgoto** subroutine should be passed to the **tputs** subroutine.

Converting termcap Descriptions to terminfo Descriptions

The **captoinfo** command converts termcap descriptions to terminfo descriptions. The following example illustrates how the **captoinfo** command works:

```
captoinfo /usr/lib/libtermcap/termcap.src
```

This command converts the **/usr/lib/libtermcap/termcap.src** file to terminfo source. The **captoinfo** command writes the output to standard output and preserves comments and other information in the file.

Manipulating TTYs

The following functions save and restore the state of terminal modes:

savetty	Saves the state of the tty modes.
resetty	Restores the state of the tty modes to what they were the last time the **savetty** subroutine was called.

Synchronous and Networked Asynchronous Terminals

Synchronous, networked synchronous (NWA) or non-standard directly-connected asynchronous terminals are often used in a mainframe environment and communicate to the host in block mode. That is, the user types characters at the terminal then presses a special key to initiate transmission of the characters to the host.

Although it may be possible to send arbitrary sized blocks to the host, it is not possible or desirable to cause a character to be transmitted with only a single keystroke. Doing so could cause severe problems to an application wishing to make use of single-character input.

Output

The curses interface can be used in the normal way for all operations pertaining to output to the terminal, with the possible exception that on some terminals the refresh( ) routine may have to redraw the entire screen contents in order to perform any update.

If it is additionally necessary to clear the screen before each such operation, the result could be undesirable.

Input

Because of the nature of operation of synchronous (block-mode) and NWA terminals, it might not be possible to support all or any of the curses input functions. In particular, the following points should be noted:

- Single-character input might not possible. It may be necessary to press a special key to cause all characters typed at the terminal to be transmitted to the host.
- It is sometimes not possible to disable echo. Character echo may be performed directly by the terminal. On terminals that behave in this way, any curses application that performs input should be aware that any characters typed will appear on the screen at wherever the cursor is positioned. This does not necessarily correspond to the position of the cursor in the window.

Working with Color

If a terminal supports color, you can use the color manipulation subroutines to include color in your curses program. Before manipulating colors, you should test whether a terminal supports color. To do this, you can use either the **has_colors** subroutine or the **can_change_color** subroutine. The **can_change_color** subroutine also checks to see if a program can change the terminal's color definitions. Neither of these subroutines requires an argument.

can_change_color	Checks to see if the terminal supports colors and changing of the color definition.
has_colors	Checks that the terminal supports colors.
start_color	Initializes the eight basic colors and two global variables, **COLORS** and **COLOR_PAIRS**.

Once you have determined that the terminal supports color, you must call the **start_color** subroutine before calling other color subroutines. It is a good practice to call this subroutine right after the **initscr** subroutine and after a successful color test. The **COLORS** global variable defines the maximum number of colors the terminal supports. The **COLOR_PAIRS** global variable defines the maximum number of color pairs the terminal supports.

Manipulating Video Attributes

Your program can manipulate a number of video attributes.

Video Attributes, Bit Masks, and the Default Colors

Curses enables you to control the following attributes:

A_STANDOUT	Terminal's best highlighting mode.
A_UNDERLINE	Underline.
A_REVERSE	Reverse video.
A_BLINK	Blinking.
A_DIM	Half-bright.
A_BOLD	Extra bright or bold.
A_ALTCHARSET	Alternate character set.
A_NORMAL	Normal attributes.
COLOR_PAIR (*Number*)	Displays the color pair represented by *Number*. You must have already initialized the color pair using the **init_pair** subroutine.

These attributes are defined in the **curses.h** file. You can pass attributes to the **wattron**, **wattroff**, and **wattrset** subroutines or you can OR them with the characters passed to the **waddch** subroutine. The C logical OR operator is a | (pipe symbol). The following bit masks are also provided:

A_NORMAL	Turns all video attributes off.
A_CHARTEXT	Extracts a character.
A_ATTRIBUTES	Extracts attributes.
A_COLOR	Extracts color-pair field information.

Two macros are provided for working with color pairs: **COLOR_PAIR(** *Number***)** and **PAIR_NUMBER(** *Attribute***)**. The **COLOR_PAIR(** *Number***)** macro and the **A_COLOR** mask are used by the **PAIR_NUMBER(** *Attribute***)** macro to extract the color-pair number found in the attributes specified by the *Attribute* parameter.

If your program uses color, the **curses.h** file defines a number of macros that identify default colors. These colors are the following:

Color	Integer Value
COLOR_BLACK	0
COLOR_BLUE	1
COLOR_GREEN	2
COLOR_CYAN	3
COLOR_RED	4
COLOR_MAGENTA	5
COLOR_YELLOW	6
COLOR_WHITE	7

Curses assumes that the default background color for all terminals is 0 (**COLOR_BLACK**).

Setting Video Attributes

The current window attributes are applied to all characters written into the window with the **addch** subroutines. These attributes remain as a property of the characters. The characters retain these attributes during terminal operations.

attroff or **wattroff**	Turns off attributes.
attron or**wattron**	Turns on attributes.
attrset or **wattrset**	Sets the current attributes of a window.
standout, **wstandout**, **standend**, or**wstandend**	Puts a window into and out of the terminal's best highlight mode.
vidputs or **vidattr**	Outputs a string that puts the terminal in a video-attribute mode.

The **attrset** subroutine sets the current attributes of the default screen. The **wattrset** subroutine sets the current attributes of the user-defined window.

Use the **attron** and **attroff** subroutines to turn on and off the specified attributes in the stdscr without affecting any others. The **wattron** and **wattroff** subroutines perform the same actions in user-defined windows.

The **standout** subroutine is the same as a call to the **attron** subroutine with the **A_STANDOUT** attribute. It puts the stdscr into the terminal's best highlight mode. The **wstandout** subroutine is the same as a call to the **wattron(** *Window***, A_STANDOUT)** subroutine. It puts the user-defined window into the terminal's best highlight mode. The **standend** subroutine is the same as a call to the **attrset(0)** subroutine. It turns off all attributes for stdscr. The **wstandend** subroutine is the same as a call to the **wattrset(** *Window***, 0)** subroutine. It turns off all attributes for the specified window.

The **vidputs** subroutine outputs a string that puts the terminal in the specified attribute mode. Characters are output through the **putc** subroutine. The **vidattr** subroutine is the same as the **vidputs** subroutine except that characters are output through the **putchar** subroutine.

Working with Color Pairs

The **COLOR_PAIR** (*Number*) macro is defined in the **curses.h** file so you can manipulate color attributes as you would any other attributes. You must initialize a color pair with the **init_pair** subroutine before you use it. The **init_pair** subroutine has three parameters *Pair*, *Foreground*, and *Background*. The *Pair* parameter must be between 1 and **COLOR_PAIRS** -1. The *Foreground* and *Background* parameters must be between 0 and **COLORS** -1. For example, to initialize color pair 1 to a foreground of black with a background of cyan, you would use the following:

```
init_pair(1, COLOR_BLACK, COLOR_CYAN);
```

You could then set the attributes for the window as:

```
wattrset(win, COLOR_PAIR(1));
```

If you then write the string `Let's add Color` to the terminal, the string appears as black characters on a cyan background.

Extracting Attributes

You can use the results from the call to the **winch** subroutine to extract attribute information, including the color-pair number. The following example uses the value returned by a call to the **winch** subroutine with the C logical AND operator (`&`) and the **A_ATTRIBUTES** bit mask to extract the attributes assigned to the current position in the window. The results from this operation are used with the **PAIR_NUMBER** macro to extract the color-pair number, and the number `1` is printed on the screen.

```
win = newwin(10, 10, 0, 0);
init_pair(1, COLOR_RED, COLOR_YELLOW);
wattrset(win, COLOR_PAIR(1));
waddstr(win, "apple");

number = PAIR_NUMBER((mvwinch(win, 0, 0) & A_ATTRIBUTES));
wprintw(win, "%d\n", number);
wrefresh(win);
```

Lights and Whistles

The curses library provides alarm subroutines to signal the user.

beep Sounds an audible alarm on the terminal
flash Displays a visible alarm on the terminal

Setting Curses Options

All curses options are initially turned off, so it is not necessary to turn them off before calling the **endwin** subroutine. The following subroutines allow you to set various options with curses:

curs_set Sets the cursor visibility to invisible, normal, or very visible.
idlok Specifies whether curses can use the hardware insert and delete line features of terminals so equipped.
intrflush Specifies whether an interrupt key (interrupt, quit, or suspend) flushes all output in the tty driver. This option's default is inherited from the tty driver.
keypad Specifies whether curses retrieves the information from the terminal's keypad. If enabled, the user can press a function key (such as an arrow key) and the **wgetch** subroutine returns a single value representing that function key. If disabled, curses will not treat the function keys specially and your program must interpret the escape sequences. For a list of these function keys, see the **wgetch** subroutine.
typeahead Instructs curses to check for type ahead in an alternative file descriptor.

See the **wgetch** subroutines and “Setting Terminal Input and Output Modes” on page 20 for descriptions of additional curses options.

Manipulating Soft Labels

Curses provides subroutines for manipulating soft function-key labels. These labels appear at the bottom of the screen and give applications, such as editors, a more user-friendly look. To use soft labels, you must call the **slk_init** subroutine before calling the **initscr** or **newterm** subroutines.

slk_clear	Clears soft labels from the screen.
slk_init	Initializes soft function key labels.
slk_label	Returns the current label.
slk_noutrefresh	Refreshs soft labels. This subroutine is functionally equivalent to the **wnoutrefresh** subroutine.
slk_refresh	Refreshs soft labels. This subroutine is functionally equivalent to the **refresh** subroutine.
slk_restore	Restores the soft labels to the screen after a call to the **slk_clear** subroutine.
slk_set	Sets a soft label.
slk_touch	Updates soft labels on the next call to the **slk_noutrefresh** subroutine.

To manage soft labels, curses reduces the size of the **stdscr** by one line. It reserves this line for use by the soft-label functions. This reservation means that the environment variable **LINES** is also reduced. Many terminals support built-in soft labels. If built-in soft labels are supported, curses uses them. Otherwise, curses simulates the soft-labels with software.

Because many terminals that support soft labels have 8 labels, curses follows the same standard. A label string is restricted to 8 characters. Curses arranges labels in one of two patterns: 3-2-3 (3 left, 2 center, 3 right) or 4-4 (4 left, 4 right).

To specify a string for a particular label, call the **slk_set** subroutine. This subroutine also instructs curses as to left-justify, right-justify, or center the string on the label. If you wish to obtain a label name before it was justified by the **slk_set** subroutine, use the **slk_label** subroutine. The **slk_clear** and **slk_restore** subroutines clear and restore soft labels respectively. Normally, to update soft labels, your program should call the **slk_noutrefresh** subroutine for each label and then use a single call to the **slk_refresh** subroutine to perform the actual output. To output all the soft labels on the next call to the **slk_noutrefresh** subroutine, use the **slk_touch** subroutine.

Obsolete Curses Subroutines

Several curses subroutines are obsolete beginning in AIX Version 4. These obsolete subroutines are emulated as indicated in the following list:

Obsolete	**Replaced by**
crmode	cbreak
fixterm	reset_prog_mode
getcap	tgetstr
nocrmode	nocbreak
resetterm	reset_shell_mode
saveterm	def_prog_mode
setterm	setupterm

The **touchoverlap**, **flushok**, and **_showstring** subroutines are obsolete and there are no direct replacements. The **gettmode** subroutine is available as a no-op.

AIX 3.2 Curses Compatibility

- In AIX 4.3, curses is not compatible with AT&T System V Release 3.2 curses.
- In versions prior to AIX 4.3, curses is compatible with AT&T System V Release 3.2 curses.
- In versions prior to AIX 4.3 curses have been kept in a form useful for supporting existing binaries only. This new change was made to provide support for color and to increase application portability to AIX systems.
- Applications already running under AIX 4.3 will not operate using the old curses.
- Applications compiled, rebound, or relinked may need source code changes for compatibility with the AIX Version 4 of curses. The newer curses library does not have or use AIX extended curses functions.
- Applications requiring multibyte support may still compile and link with extended curses. However, because the extended curses library may be removed in the future, use of the extended curses library is discouraged except for applications that require multibyte support.

List of Additional Curses Subroutines

For information on the X/Open UNIX95 Specification curses subroutines available on AIX 4.2 (and later), see the X/Open CAE Specification.

Manipulating Windows

scr_dump	Writes the current contents of the virtual screen to the specified file.
scr_init	Uses the contents of a specified file to initialize the curses data structures.
scr_restore	Sets the virtual screen to the contents of the specified file.

Manipulating Characters

echochar, **wechochar**, or **pechochar**	Functionally equivalent to a call to the **addch** (or **waddch**) subroutine followed by a call to the **refresh** (or **wrefresh**) subroutine.
flushinp	Flushes any type-ahead characters typed by the user but not yet read by the program.
insertln or **winsertln**	Inserts a blank line in a window.
keyname	Returns a pointer to a character string containing a symbolic name for the *Key* parameter.
meta	Determines whether 8-bit character return for the **wgetch** subroutine is allowed.
nodelay	Causes a call to the **wgetch** subroutine to be a nonblocking call. If no input is ready, the **wgetch** subroutine returns **ERR**.
scroll	Scrolls a window up one line.
unctrl	Returns the printable representation of a character. Control characters are punctuated with a ^ (caret).
vwprintw	Performs the same operation as the **wprintw** subroutine but takes a variable list of arguments.
vwscanw	Performs the same operation as the **wscanw** subroutine but takes a variable list of arguments.

Manipulating Terminals

def_prog_mode	Identifies the current terminal mode as the in-curses mode.
def_shell_mode	Saves the current terminal mode as the not-in-curses mode.
del_curterm	Frees the space pointed to by the *oterm* variable.

notimeout	Prevents the **wgetch** subroutine from setting a timer when interpreting an input escape sequence.
pechochar	Equivalent to a call to the **waddch** subroutine followed by a call to the **prefresh** subroutine.
reset_prog_mode	Restores the terminal into the in-curses program mode.
reset_shell_mode	Restores the terminal to shell mode (out-of-curses mode). The **endwin** subroutine does this automatically.
restartterm	Sets up a TERMINAL structure for use by curses. This subroutine is similar to the **setupterm** subroutine. Call the **restartterm** subroutine after restoring memory to a previous state. For example, call this subroutine after a call to the **scr_restore** subroutine.

Manipulating Color

color_content	Returns the composition of a color.
init_color	Changes a color to the desired composition.
init_pair	Initializes a color pair to the specified foreground and background colors.
pair_content	Returns the foreground and background colors for a specified color-pair number.

Miscellaneous Utilities

baudrate	Queries the current terminal and returns its output speed.
erasechar	Returns the erase character chosen by the user.
killchar	Returns the line-kill character chosen by the user.

Chapter 3. Debugging Programs

There are several debug programs available for debugging your programs: the **adb**, **dbx**, **dex**, **softdb**, and kernel debug programs. The **adb** program enables you to debug executable binary files and examine non-ASCII data files. The **dbx** program enables source-level debugging of C, C++, Pascal, and FORTRAN language programs, as well as assembler-language debugging of executable programs at the machine level. The (**dex**) provides an X interface for the **dbx** debug program, providing windows for viewing the source, context, and variables of the application program. The **softdb** debug program works much like the **dex** debug program, but **softdb** is used with AIX Software Development Environment Workbench. The kernel debug program is used to help determine errors in code running in the kernel.

adb Debug Program Overview

The **adb** command provides a general purpose debug program. You can use this command to examine object and core files and provide a controlled environment for running a program.

While the **adb** command is running, it takes standard input and writes to standard output. The command does not recognize the Quit or Interrupt keys. If these keys are used, the **adb** command waits for a new command.

Getting Started with the adb Debug Program

This section explains how to start the **adb** debugging program from a variety of files, use the **adb** prompt, use shell commands from within the **adb** program, and stop the **adb** program.

Starting adb with a Program File

You can debug any executable C or assembly language program file by entering a command line of the form:

adb *FileName*

where *FileName* is the name of the executable program file to be debugged. The **adb** program opens the file and prepares its text (instructions) and data for subsequent debugging. For example, the command:

```
adb sample
```

prepares the program named `sample` for examination and operation.

Once started, the **adb** debug program places the cursor on a new line and waits for you to type commands.

Starting adb with a Nonexistent or Incorrect File

If you start the **adb** debug program with the name of a nonexistent or incorrectly formatted file, the **adb** program first displays an error message and then waits for commands. For example, if you start the **adb** program with the command:

```
adb sample
```

and the `sample` file does not exist, the **adb** program displays the message:

```
sample: no such file or directory.
```

Starting adb with the Default File

You can start the **adb** debug program without a file name. In this case, the **adb** program searches for the default **a.out** file in your current working directory and prepares it for debugging. Thus, the command:

```
adb
```

is the same as entering:

```
adb a.out
```

The **adb** program starts with the **a.out** file and waits for a command. If the **a.out** file does not exist, the **adb** program starts without a file and does not display an error message.

Starting adb with a Core Image File

You can use the **adb** debug program to examine the core image files of programs that caused irrecoverable system errors. Core image files maintain a record of the contents of the CPU registers, stack, and memory areas of your program at the time of the error. Therefore, core image files provide a way to determine the cause of an error.

To examine a core image file with its corresponding program, you must give the name of both the core and the program file. The command line has the form:

adb *ProgramFile CoreFile*

where *ProgramFile* is the file name of the program that caused the error, and *CoreFile* is the file name of the core image file generated by the system. The **adb** program then uses information from both files to provide responses to your commands.

If you do not give the filename of the core image file, the **adb** program searches for the default core file, named **core**, in your current working directory. If such a file is found, the **adb** program determines whether the core file belongs to the *ProgramFile*. If so, the **adb** program uses it. Otherwise, the **adb** program discards the core file by giving an appropriate error message.

> **Note:** The **adb** command cannot be used to examine 64-bit objects and AIX 4.3 core format. **adb** still works with pre-AIX 4.3 core format. On AIX 4.3, user can make kernel to generate pre-AIX 4.3 style core dumps using smitty.

Starting adb with a Data File

The **adb** program provides a way to look at the contents of the file in a variety of formats and structures. You can use the **adb** program to examine data files by giving the name of the data file in place of the program or core file. For example, to examine a data file named **outdata**, enter:

```
adb outdata
```

The **adb** program opens a file called `outdata` and lets you examine its contents. This method of examining files is useful if the file contains non-ASCII data. The **adb** command may display a warning when you give the name of a non-ASCII data file in place of a program file. This usually happens when the content of the data file is similar to a program file. Like core files, data files cannot be executed.

Starting adb with the Write Option

If you open a program or data file with the **-w** flag of the **adb** command, you can make changes and corrections to the file. For example, the command:

```
adb -w sample
```

opens the program file `sample` for writing. You can then use **adb** commands to examine and modify this file. The **-w** flag causes the **adb** program to create a given file if it does not already exist. The option also lets you write directly to memory after running the given program.

Using a Prompt

After you have started the **adb** program you can redefine your prompt with the **$P** subcommand.

To change the **[adb:scat]>>** prompt to **Enter a debug command—->**, enter:

```
$P"Enter a debug command--->"
```

The quotes are not necessary when redefining the new prompt from the **adb** command line.

Using Shell Commands from within the adb Program

You can run shell commands without leaving the **adb** program by using the **adb** escape command (**!**) (exclamation point). The escape command has the form:

! *Command*

In this format *Command* is the shell command you want to run. You must provide any required arguments with the command. The **adb** program passes this command to the system shell that calls it. When the command is finished, the shell returns control to the **adb** program. For example, to display the date, enter the following command:

```
! date
```

The system displays the date and restores control to the **adb** program.

Exiting the adb Debug Program

You can stop the **adb** program and return to the system shell by using the **$q** or **$Q** subcommands. You can also stop the **adb** program by typing the Ctrl-D key sequence. You cannot stop the **adb** program by pressing the Interrupt or Quit keys. These keys cause **adb** to wait for a new command. For more information, see (“Stopping a Program with the Interrupt and Quit Keys” on page 36).

Controlling Program Execution

This section explains the commands and subcommands necessary to prepare programs for debugging; execute programs; set, display, and delete breakpoints; continue programs; single-step through a program; stop programs; and kill programs.

Preparing Programs for Debugging with the adb Program

Compile the program using the **cc** command to a file such as **adbsamp2** by entering the following:

```
cc adbsamp2.c -o adbsamp2
```

To start the debug session, enter:

```
adb adbsamp2
```

The C language does not generate statement labels for programs. Therefore, you cannot refer to individual C language statements when using the debug program. To use execution commands effectively, you must be familiar with the instructions that the C compiler generates and how those instructions relate to individual C language statements. One useful technique is to create an assembler language listing of your

C program before using the **adb** program. Then, refer to the listing as you use the debug program. To create an assembler language listing, use the **-S** or **-q***List* flag of the **cc** command.

For example, to create an assembler language listing of the example program, **adbsamp2.c**, use the following command:

```
cc -S adbsamp2.c -o adbsamp2
```

This command creates the **adbsamp2.s** file, that contains the assembler language listing for the program, and compiles the program to the executable file, **adbsamp2**.

Running a Program

You can execute a program by using the **:r** or **:R** subcommand. For more information see, ("Displaying and Manipulating the Source File with the adb Program" on page 43). The commands have the form:

[*Address*][**,***Count*] **:r** [*Arguments*]

OR

[*Address*][**,***Count*] **:R** [*Arguments*]

In this format, the *Address* parameter gives the address at which to start running the program; the *Count* parameter is the number of breakpoints to skip before one is taken; and the *Arguments* parameter provides the command-line arguments, such as file names and options, to pass to the program.

If you do not supply an *Address* value, the **adb** program uses the start of the program. To run the program from the beginning enter:

```
:r
```

If you supply a *Count* value, the **adb** program ignores all breakpoints until the given number has been encountered. For example, to skip the first five named breakpoints, use the command:

```
,5:r
```

If you provide arguments, separate them by at least one space each. The arguments are passed to the program in the same way the system shell passes command-line arguments to a program. You can use the shell redirection symbols.

The **:R** subcommand passes the command arguments through the shell before starting program operation. You can use shell pattern-matching characters in the arguments to refer to multiple files or other input values. The shell expands arguments containing pattern-matching characters before passing them to the program. This feature is useful if the program expects multiple file names. For example, the following command passes the argument **[a-z]*** to the shell where it is expanded to a list of the corresponding file names before being passed to the program:

```
:R [a-z]*.s
```

The **:r** and **:R** subcommands remove the contents of all registers and destroy the current stack before starting the program. This operation halts any previous copy of the program that may be running.

Setting Breakpoints

To set a breakpoint in a program, use the **:b** subcommand. Breakpoints stop operation when the program reaches the specified address. Control then returns to the **adb** debug program. The command has the form:

[*Address*] [,*Count*] **:b** [*Command*]

In this format, the *Address* parameter must be a valid instruction address; the *Count* parameter is a count of the number of times you want the breakpoint to be skipped before it causes the program to stop; and the *Command* parameter is the **adb** command you want to execute each time that the instruction is executed (regardless of whether the breakpoint stops the program). If the specified command sets . (period) to a value of 0, the breakpoint causes a stop.

Set breakpoints to stop program execution at a specific place in the program, such as the beginning of a function, so that you can look at the contents of registers and memory. For example, when debugging the example **adbsamp2** program, the following command sets a breakpoint at the start of the function named **f**:

```
.f :b
```

The breakpoint is taken just as control enters the function and before the function's stack frame is created.

A breakpoint with a count is used within a function that is called several times during the operation of a program, or within the instructions that correspond to a **for** or **while** statement. Such a breakpoint allows the program to continue to run until the given function or instructions have been executed the specified number of times. For example, the following command sets a breakpoint for the second time that the **f** function is called in the **adbsamp2** program:

```
.f,2 :b
```

The breakpoint does not stop the function until the second time the function is run.

Displaying Breakpoints

Use the **$b** subcommand to display the location and count of each currently defined breakpoint. This command displays a list of the breakpoints by address and any count or commands specified for the breakpoints. For example, the following sets two breakpoints in the **adbsamp2** file and then uses the **$b** subcommand to display those breakpoints:

```
.f+4:b
.f+8:b$v
$b
breakpoints
count  brkpt            command
1      .f+8             $v
1      .f+4
```

When the program runs, it stops at the first breakpoint that it finds, such as `.f+4`. If you use the **:c** subcommand to continue execution, the program stops again at the next breakpoint and starts the **$v** subcommand. The command and response sequence looks like the following example:

```
:r
adbsamp2:running
breakpoint        .f+4:         st       r3,32(r1)
:c
adbsamp2:running
variables
b = 268435456
d = 236
e = 268435512
m = 264
breakpoint        .f+8           l       r15,32(r1)
```

Deleting Breakpoints

To use the **:d** subcommand to delete a breakpoint from a program, enter:

Address **:d**

In this format, the *Address* parameter gives the address of the breakpoint to delete.

For example, when debugging the example **adbsamp2** program, entering the following command deletes the breakpoint at the start of the **f** function:

```
.f:d
```

Continuing Program Execution

To use the **:c** subcommand to continue the execution of a program after it has been stopped by a breakpoint enter:

[*Address*] [*,Count*] **:c** [*Signal*]

In this format, the *Address* parameter gives the address of the instruction at which to continue operation; the *Count* parameter gives the number of breakpoints to ignore; and the *Signal* parameter is the number of the signal to send to the program.

If you do not supply an *Address* parameter, the program starts at the next instruction after the breakpoint. If you supply a *Count* parameter, the **adb** debug program ignores the first *Count* breakpoints.

If the program is stopped using the Interrupt or Quit key, this signal is automatically passed to the program upon restarting. To prevent this signal from being passed, enter the command in the form:

[*Address*] [,*Count*] **:c 0**

The command argument **0** prevents a signal from being sent to the subprocess.

Single-Stepping a Program

Use the **:s** subcommand to run a program in single steps or one instruction at a time. This command issues an instruction and returns control to the **adb** debug program. The command has the form:

[A*address*] [,*Count*] **:s** [*Signal*]

In this format, the *Address* parameter gives the address of the instruction you want to execute, and the *Count* parameter is the number of times you want to repeat the command. If there is no current subprocess, the *ObjectFile* parameter is run as a subprocess. In this case, no signal can be sent and the remainder of the line is treated as arguments to the subprocess. If you do not supply a value for the *Address* parameter, the **adb** program uses the current address. If you supply the *Count* parameter, the **adb** program continues to issue each successive instruction until the *Count* parameter instructions have been run. Breakpoints are ignored while single-stepping. For example, the following command issues the first five instructions in the **main** function:

```
.main,5:s
```

Stopping a Program with the Interrupt and Quit Keys

Use either the Interrupt or Quit key to stop running a program at any time. Pressing either of these keys stops the current program and returns control to the **adb** program. These keys are useful with programs that have infinite loops or other program errors.

When you press the Interrupt or Quit key to stop a program, the **adb** program automatically saves the signal. If you start the program again using the **:c** command, the **adb** program automatically passes the signal to the program. This feature is useful when testing a program that uses these signals as part of its processing. To continue running the program without sending signals, use the command:

```
:c 0
```

The command argument **0** (zero) prevents a signal from being sent to the program.

Stopping a Program

To stop a program you are debugging, use the **:k** subcommand. This command stops the process created for the program and returns control to the **adb** debug program. The command clears the current contents of the system unit registers and stack and begins the program again. The following example shows the use of the **:k** subcommand to clear the current process from the **adb** program:

```
:k
560:    killed
```

Using adb Expressions

This section describes the use of **adb** expressions.

Using Integers in Expressions

When creating an expression, you can use integers in three forms: decimal, octal, and hexadecimal. Decimal integers must begin with a non-zero decimal digit. Octal numbers must begin with a 0 (zero) and have octal digits only (0-7). Hexadecimal numbers must begin with the prefix 0x and can contain decimal digits and the letters a through f (in both uppercase and lowercase). The following are examples of valid numbers:

Decimal	Octal	Hexadecimal
34	042	0x22
4090	07772	0xffa

Using Symbols in Expressions

Symbols are the names of global variables and functions defined within the program being debugged. Symbols are equal to the address of the given variable or function. They are stored in the program symbol table and are available if the symbol table has not been stripped from the program file.

In expressions, you can spell the symbol exactly as it is in the source program or as it has been stored in the symbol table. Symbols in the symbol table are no more than 8 characters long.

When you use the **?** subcommand, the **adb** program uses the symbols found in the symbol table of the program file to create symbolic addresses. Thus, the **?** subcommand sometimes gives a function name when displaying data. This does not happen if the **?** subcommand is used for text (instructions) and the **/** command is used for data.

Local variables can only be addressed if the C language source program is compiled with the **-g** flag.

If the C language source program is not compiled using the **-g** flag the local variable cannot be addressed. The following command displays the value of the local variable **b** in a function sample:

```
.sample.b / x - value of local variable.
.sample.b = x - Address of local variable.
```

Using Operators in Expressions

You can combine integers, symbols, variables, and register names with the following operators:

Unary Operators:

~ (tilde)	Bitwise complementation
- (dash)	Integer negation
***** (asterisk)	Returns contents of location

Binary Operators:	
+ (plus)	Addition
- (minus)	Subtraction
***** (asterisk)	Multiplication
% (percent)	Integer division
& (ampersand)	Bitwise conjunction
] (right bracket)	Bitwise disjunction
^ (caret)	Modulo
# (number sign)	Round up to the next multiple

The **adb** debug program uses 32-bit arithmetic. Values that exceed 2,147,483,647 (decimal) are displayed as negative values. The following example shows the results of assigning two different values to the variable *n*, and then displaying the value in both decimal and hexadecimal:

```
2147483647>n<
n=D
    2147483647<
n=X
    7fffffff
2147483648>n<
n=D
    -2147483648<
n=X
    80000000
```

Unary operators have higher precedence than binary operators. All binary operators have the same precedence and are evaluated in order from left to right. Thus, the **adb** program evaluates the following binary expressions as shown:

```
2*3+4=d
    10
4+2*3=d
    18
```

You can change the precedence of the operations in an expression by using parentheses. The following example shows how the previous expression is changed by using parentheses:

```
4+(2*3)=d
    10
```

The unary operator, ***** (asterisk), treats the given address as a pointer into the data segment. An expression using this operator is equal to the value pointed to by that pointer. For example, the expression:

```
*0x1234
```

is equal to the value at the data address 0x1234, whereas the example:

```
0x1234
```

is equal to 0x1234.

Customizing the adb Debug Program

This section describes how you can customize the **adb** debug program.

Combining Commands on a Single Line

You can give more than one command on a line by separating the commands with a ; (semicolon). The commands are performed one at a time, starting at the left. Changes to the current address and format carry over to the next command. If an error occurs, the remaining commands are ignored. For example, the following sequence displays both the **adb** variables and then the active subroutines at one point in the **adbsamp2** program:

```
$v;$c
variables
b = 10000000
d = ec
e = 10000038
m = 108
t = 2f8.
f(0,0) .main+26.
main(0,0,0) start+fa
```

Creating adb Scripts

You can direct the **adb** debug program to read commands from a text file instead of from the keyboard by redirecting the standard input file when you start the **adb** program. To redirect standard input, use the input redirection symbol, < (less than), and supply a file name. For example, use the following command to read commands from the file script:

```
adb sample <script
```

The file must contain valid **adb** subcommands. Use the **adb** program script files when the same set of commands can be used for several different object files. Scripts can display the contents of core files after a program error. The following example shows a file containing commands that display information about a program error. When that file is used as input to the **adb** program using the following command to debug the **adbsamp2** file, the specified output is produced.

```
120$w
4095$s.
f:b:
r
=ln"======= adb Variables ======="
$v
=ln"======= Address Map ======="
$m
=ln"======= C Stack Backtrace ======="
$C
=ln"======= C External Variables ======="
$e
=ln"======= Registers ======="
$r
0$s
=ln"======= Data Segment =======">
b,10/8xna
```

```
$ adb adbsamp2 <script
```

```
adbsamp2: running
breakpoint .f:   b  .f+24
    ======= adb Variables =======
variables
0 = TBD
1 = TBD
2 = TBD
9 = TBD
b = 10000000
d = ec
e = 10000038
m = 108
```

```
t = 2f8
    ======= Address Map =======
[0]? map  .adbsamp2.
b1 = 10000000  e1 = 100002f8  f1 = 0
b2 = 200002f8  e2 = 200003e4  f2 = 2f8
[0]/ map  .-.
b1 = 0    e1 = 0    f1 = 0
b2 = 0    e2 = 0    f2 = 0
    ======= C Stack Backtrace =======.
f(0,0) .main+26.
main(0,0,0) start+fa
    ======= C External Variables =======Full word.
errno: 0.
environ:  3fffe6bc.
NLinit:  10000238.
main: 100001ea.
exit: 1000028c.
fcnt: 0

.loop .count:  1.
f:   100001b4.
NLgetfile: 10000280.
write: 100002e0.
NLinit. .X:  10000238 .
NLgetfile. .X:   10000280 .
cleanup: 100002bc.
exit: 100002c8 .
exit . .X:  1000028c . .
cleanup . .X:  100002bc

    ======= Registers =======
mq  20003a24  .errno+3634
cs  100000 gt
ics  1000004
pc  100001b4 .f
r15  10000210 .main+26
r14  20000388  .main
r13  200003ec  .loop .count
r12  3fffe3d0
r11  3fffe44c
r10  0
r9  20004bcc
r8  200041d8  .errno+3de8
r7  0
r6  200030bc  .errno+2ccc
r5  1
r4  200003ec  .loop .count
r3  f4240
r2  1
r1  3fffe678
r0  20000380  .f.
f:   b  .f+24

    ======= Data Segment =======
10000000:  103 5313  3800  0  0  2f8 0  ec
10000010:  0  10 1000  38 0  0  0  1f0
10000020:  0  0  0  0  1000  0  2000  2f8
10000030:  0  0  0  0  4  6000  0  6000
10000040:  6e10  61d0  9430  a67 6730  6820  c82e  8
10000050:  8df0  94 cd0e  60 6520  a424  a432  c84e
10000060:  8  8df0  77 cd0e  64 6270  8df0  86
10000070:  cd0e  60 6520  a424  a432  6470  8df0  6a
10000080:  cd0e  64 c82e  19 8df0  78 cd0e  60
10000090:  6520  a424  a432  c84e  19 8df0  5b cd0e
100000a0:  64 cd2e  5c 7022  d408  64 911 c82e
100000b0:  2e 8df0  63 cd0e  60 6520  a424  a432
100000c0:  c84e  2e 8df0  46 cd0e  64 15 6280
```

```
100000d0:  8df0  60 cd0e  68 c82e  3f 8df0  4e
100000e0:  cd0e  60 6520  a424  a432  c84e  3f 8df0
100000f0:  31 cd0e  64 c820  14 8df0  2b cd0e
10000100:
```

Setting Output Width

Use the **$w** subcommand to set the maximum width (in characters) of each line of output created by the **adb** program. The command has the form:

*Width***$w**

In this format, the *Width* parameter is an integer that specifies the width in characters of the display. You can give any width convenient for your display device. When the **adb** program is first invoked, the default width is 80 characters.

This command can be used when redirecting output to a line printer or special output device. For example, the following command sets the display width to 120 characters, a common maximum width for line printers:

```
120$w
```

Setting the Maximum Offset

The **adb** debug program normally displays memory and file addresses as the sum of a symbol and an offset. This format helps to associate the instructions and data on the display with a particular function or variable. When the **adb** program starts up, it sets the maximum offset to 255, so that symbolic addresses are assigned only to instructions or data that occur less than 256 bytes from the start of the function or variable. Instructions or data beyond that point are given numeric addresses.

In many programs, the size of a function or variable is actually larger than 255 bytes. For this reason the **adb** program lets you change the maximum offset to accommodate larger programs. You can change the maximum offset by using the **$s** subcommand.

The subcommand has the form:

*Offset***$s**

In this format, the *Offset* parameter is an integer that specifies the new offset. For example, the following command increases the maximum possible offset to 4095:

```
4095$s
```

All instructions and data that are less than 4096 bytes away are given symbolic addresses. You can disable all symbolic addressing by setting the maximum offset to zero. All addresses are given numeric values instead.

Setting Default Input Format

To alter the default format for numbers used in commands, use the **$d** or **$o** (octal) subcommands. The default format tells the **adb** debug program how to interpret numbers that do not begin with 0 (octal) or 0x (hexadecimal), and how to display numbers when no specific format is given. Use these commands to work with a combination of decimal, octal, and hexadecimal numbers.

The **$o** subcommand sets the radix to 8 and thus sets the default format for numbers used in commands to octal. After you enter that subcommand, the **adb** program displays all numbers in octal format except those specified in some other format.

The format for the **$d** subcommand is the *Radix***$d** command, where the *Radix* parameter is the new value of the radix. If the *Radix* parameter is not specified, the **$d** subcommand sets the radix to a default value of 16. When you first start the **adb** program, the default format is hexadecimal. If you change the default format, you can restore it as necessary by entering the **$d** subcommand by itself:

```
$d
```

To set the default format to decimal, use the following command:

```
0xa$d
```

Changing the Disassembly Mode

Use the **$i** and **$n** subcommands to force the **adb** debug program to disassemble instructions using the specified instruction set and mnemonics. The **$i** subcommand specifies the instruction set to be used for disassembly. The **$n** subcommand specifies the mnemonics to be used in disassembly.

If no value is entered, these commands display the current settings.

The **$i** subcommand accepts the following values:

com Specifies the instruction set for the common intersection mode of the PowerPC and POWER family.
pwr Specifies the instruction set and mnemonics for the POWER-based platform implementation of the POWER family.
pwrx Specifies the instruction set and mnemonics for the POWER2 implementation of the POWER family.
ppc Specifies the instruction set and mnemonics for the PowerPC.
601 Specifies the instruction set and mnemonics for the PowerPC 601 RISC Microprocessor.

603 Specifies the instruction set and mnemonics for the PowerPC 603 RISC Microprocessor.
604 Specifies the instruction set and mnemonics for the PowerPC 604 RISC Microprocessor.
ANY Specifies any valid instruction. For instruction sets that overlap, the mnemonics will default to POWER-based platform mnemonics.

The **$n** subcommand accepts the following values:

pwr Specifies the mnemonics for the POWER-based implementation of the POWER family.
ppc Specifies the mnemonics for the POWER-based platform.

Computing Numbers and Displaying Text

You can perform arithmetic calculations while in the **adb** debug program by using the **=** (equal sign) subcommand. This command directs the **adb** program to display the value of an expression in a specified format. The command converts numbers in one base to another, double-checks the arithmetic performed by a program, and displays complex addresses in simpler form. For example, the following command displays the hexadecimal number 0x2a as the decimal number 42:

```
0x2a=d
        42
```

Similarly, the following command displays 0x2a as the ASCII character * (asterisk):

```
0x2a=c
        *
```

Expressions in a command can have any combination of symbols and operators. For example, the following command computes a value using the contents of the **r0** and **r1** registers and the **adb** variable **b**.

```
<r0-12*<r1+<b+5=X
    8fa86f95
```

You can also compute the value of external symbols to check the hexadecimal value of an external symbol address, by entering:

```
main+5=X
    2000038d
```

The **=** (equal sign) subcommand can also display literal strings. Use this feature in the **adb** program scripts to display comments about the script as it performs its commands. For example, the following subcommand creates three lines of spaces and then prints the message `C Stack Backtrace`:

```
=3n"C Stack Backtrace"
```

Displaying and Manipulating the Source File with the adb Program

The following sections describe how you can use the **adb** program to display and manipulate the source file.

Displaying Instructions and Data

The **adb** program provides several subcommands for displaying the instructions and data of a given program and the data of a given data file. The subcommands and their formats are:

Display address	*Address* [, *Count*] **=** *Format*
Display instruction	*Address* [, *Count*] **?** *Format*
Display value of variable	*Address* [, *Count*] **/** *Format*

In this format, the symbols and variables have the following meaning:

Address	Gives the location of the instruction or data item.
Count	Gives the number of items to be displayed.
Format	Defines how to display the items.
=	Displays the address of an item.
?	Displays the instructions in a text segment.
/	Displays the value of variables.

Forming Addresses

In the **adb** program addresses are 32-bit values that indicate a specific memory address. They can, however, be represented in the following forms:

Absolute address	The 32-bit value is represented by an 8-digit hexadecimal number, or its equivalent in one of the other number-base systems.
Symbol name	The location of a symbol defined in the program can be represented by the name of that symbol in the program.
Entry points	The entry point to a routine is represented by the name of the routine preceded by a . (period). For example, to refer to the address of the start of the `main` routine, use the following notation: `.main`
Displacements	Other points in the program can be referred to by using displacements from entry points in the program. For example, the following notation references the instruction that is 4 bytes past the entry point for the symbol `main`: `.main+4`

Displaying an Address

Use the **=** (equal sign) subcommand to display an address in a given format. This command displays instruction and data addresses in a simpler form and can display the results of arithmetic expressions. For example, entering:

```
main=an
```

displays the address of the symbol `main`:

```
10000370:
```

The following example shows a command that displays (in decimal) the sum of the internal variable **b** and the hexadecimal value 0x2000, together with its output:

```
<b+0x2000=D
    268443648
```

If a count is given, the same value is repeated that number of times. The following example shows a command that displays the value of `main` twice and the output that it produces:

```
main,2=x
    370 370
```

If no address is given, the current address is used. After running the above command once (setting the current address to `main`), the following command repeats that function:

```
,2=x
    370 370
```

If you do not specify a format, the **adb** debug program uses the last format that was used with this command. For example, in the following sequence of commands, both `main` and `one` are displayed in hexadecimal:

```
main=x
  370
one=
  33c
```

Displaying the C Stack Backtrace

To trace the path of all active functions, use the **$c** subcommand. This subcommand lists the names of all functions that have been called and have not yet returned control. It also lists the address from which each function was called and the arguments passed to each function. For example, the following command sequence sets a breakpoint at the function address `.f+2` in the **adbsamp2** program. The breakpoint calls the **$c** subcommand. The program is started, runs to the breakpoint, and then displays a backtrace of the called C language functions:

```
.f+2:b$c
:r
adbsamp2:running
.f(0,0) .main+26
.main(0,0,0) start+fa
breakpoint                f+2:           tgte     r2,r2
```

By default, the **$c** subcommand displays all calls. To display fewer calls, supply a count of the number of calls to display. For example, the following command displays only one of the active functions at the preceding breakpoint:

```
,1$c
.f(0,0) .main+26
```

Choosing Data Formats

A *format* is a letter or character that defines how data is to be displayed. The following are the most commonly used formats:

Letter	Format
a	The current symbolic address
b	One byte in octal (displays data associated with instructions, or the high or low byte of a register)
c	One byte as a character (char variables)
d	Halfword in decimal (short variables)
D	Fullword in decimal (long variables)
i	Machine instructions in mnemonic format
n	A new line
o	Halfword in octal (short variables)
O	Fullword in octal (long variables)
r	A blank space
s	A null-terminated character string (null-terminated arrays of char variables)
t	A horizontal tab
u	Halfword as an unsigned integer (short variables)
x	Halfword in hexadecimal (short variables)
X	Fullword in hexadecimal (long variables)

For example, the following commands produce the indicated output when using the **adbsamp** example program:

Command	Response
main=o	1560
main=O	4000001560
main=d	880
main=D	536871792
main=x	370
main=X	20000370
main=u	880

A format can be used by itself or combined with other formats to present a combination of data in different forms. You can combine the **a**, **n**, **r**, and **t** formats with other formats to make the display more readable.

Changing the Memory Map

You can change the values of a memory map by using the **?m** and **/m** subcommands. See, (“adb Debug Program Reference Information” on page 49). These commands assign specified values to the corresponding map entries. The commands have the form:

```
[,count] ?m b1 e1 f1
[,count] /m b1 e1 f2
```

The following example shows the results of these commands on the memory map displayed with the **$m** subcommand in the previous example:

```
,0?m    10000100         10000470       0
/m      100      100     100
$m
  [0] :   ?map :   'adbsamp3'
 b1 = 0x10000100,  e1 = 10000470,  f1 = 0
 b2 = 0x20000600,  e2 = 0x2002c8a4, f2 = 0x600

 [1] :  ?map :   'shr.o' in library '/usr/ccs/lib/libc.a'
```

```
 b1 = 0xd00d6200,  e1 = 0xd01397bf,  f1 = 0xd00defbc
 b2 = 0x20000600,  e2 = 0x2002beb8, f2 = 0x4a36c

  [-] : /map :  '-'
 b1 = 100,    e1 = 100,   f1 = 100
 b2 = 0,    e2 = 0,     f2 = 0
```

To change the data segment values, add an ***** (asterisk) after the **/** or **?**.

```
,0?*m   20000270         20000374       270
/*m     200      200     200
$m
  [0] : ?map :   'adbsamp3'
 b1 = 0x10000100,  e1 = 10000470,  f1 = 0
 b2 = 0x20000270,  e2 = 0x20000374, f2 = 0x270

 [1] :  ?map :   'shr.o' in library '/usr/ccs/lib/libc.a'
 b1 = 0xd00d6200,  e1 = 0xd01397bf,  f1 = 0xd00defbc
 b2 = 0x20000600,  e2 = 0x2002beb8, f2 = 0x4a36c

  [-] :  /map :  '-'
 b1 = 100,    e1 = 100,   f1 = 100
 b2 = 0,    e2 = 0,     f2 = 0
```

Patching Binary Files

You can make corrections or changes to any file, including executable binary files, by starting the **adb** program with the **-w** flag and by using the **w** and **W** (“adb Debug Program Reference Information” on page 49) subcommands.

Locating Values in a File

Locate specific values in a file by using the **l** and **L** subcommands. See (“adb Debug Program Reference Information” on page 49). The subcommands have the form:

?l *Value*

OR

/l *Value*

The search starts at the current address and looks for the expression indicated by *Value*. The **l** subcommand searches for 2-byte values. The **L** subcommand searches for 4-byte values.

The **?l** subcommand starts the search at the current address and continues until the first match or the end of the file. If the value is found, the current address is set to that value’s address. For example, the following command searches for the first occurrence of the **f** symbol in the **adbsamp2** file:

```
?l .f.
write+a2
```

The value is found at **.write+a2** and the current address is set to that address.

Writing to a File

Write to a file by using the **w** and **W** subcommands. See (“adb Debug Program Reference Information” on page 49). The subcommands have the form:

[*Address*] **?w** *Value*

In this format, the *Address* parameter is the address of the value you want to change, and the *Value* parameter is the new value. The **w** subcommand writes 2-byte values. The **W** subcommand writes 4-byte values. For example, the following commands change the word ″This″ to ″The″:

```
?l .Th.
?W .The.
```

The **W** subcommand changes all four characters.

Making Changes to Memory

Make changes to memory whenever a program has run. If you have used an **:r** subcommand with a breakpoint to start program operation, subsequent **w** subcommands cause the **adb** program to write to the program in memory rather than to the file. This command is used to make changes to a program's data as it runs, such as temporarily changing the value of program flags or variables.

Using adb Variables

The **adb** debug program automatically creates a set of its own variables when it starts. These variables are set to the addresses and sizes of various parts of the program file as defined in the following table:

Variable	Content
0	Last value printed
1	Last displacement part of an instruction source
2	Previous value of the **1** variable
9	Count on the last **$<** or **$<<** command
b	Base address of the data segment
d	Size of the data segment
e	Entry address of the program
m	″Magic″ number
s	Size of the stack segment
t	Size of the text segment

The **adb** debug program reads the program file to find the values for these variables. If the file does not seem to be a program file, then the **adb** program leaves the values undefined.

To display the values that the **adb** debug program assigns to these variables, use the **$v** subcommand. For more information, see (“adb Debug Program Reference Information” on page 49). This subcommand lists the variable names followed by their values in the current format. The subcommand displays any variable whose value is not 0 (zero). If a variable also has a non-zero segment value, the variable's value is displayed as an address. Otherwise, it is displayed as a number. The following example shows the use of this command to display the variable values for the sample program **adbsamp**:

```
$v
Variables
0 = undefined
1 = undefined
2 = undefined
9 = undefined
b = 10000000
d = 130
e = 10000038
m = 108
t = 298
```

Specify the current value of an **adb** variable in an expression by preceding the variable name with < (less than sign). The following example displays the current value of the **b** base variable:

```
<b=X
10000000
```

Create your own variables or change the value of an existing variable by assigning a value to a variable name with > (greater than sign). The assignment has the form:

Expression > *VariableName*

where the *Expression* parameter is the value to be assigned to the variable and the *VariableName* parameter is the variable to receive the value. The *VariableName* parameter must be a single letter. For example, the assignment:

```
0x2000>b
```

assigns the hexadecimal value 0x2000 to the **b** variable. Display the contents of **b** again to show that the assignment occurred:

```
<b=X
 2000
```

Finding the Current Address

The **adb** program has two special variables that keep track of the last address used in a command and the last address typed with a command. The **.** (period) variable, also called the current address, contains the last address used in a command. The ″ (double quotation mark) variable contains the last address typed with a command. The **.** and ″ variables usually contain the same address except when implied commands, such as the newline and ^ (caret) characters, are used. These characters automatically increase and decrease the **.** variable but leave the) (right parenthesis) variable unchanged.

Both the **.** and the ″ variables can be used in any expression. The < (less than sign) is not required. For example, the following commands display these variables at the start of debugging with the **adbsamp** (“Example adb Program: adbsamp” on page 54) program:

```
.=
    0.
=
    0
```

Displaying External Variables

Use the **$e** (“adb Debug Program Reference Information” on page 49) subcommand to display the values of all external variables in the **adb** program. External variables are the variables in your program that have global scope or have been defined outside of any function, and include variables defined in library routines used by your program, as well as all external variables of shared libraries.

The **$e** subcommand is useful to get a list of the names for all available variables or a summary of their values. The command displays one name on each line with the variable’s value (if any) on the same line. If the *Count* parameter is specified, only the external variables associated with that file are printed.

The following example illustrates the setting of a breakpoint in the **adbsamp2** (“Example adb Program: adbsamp2” on page 55) sample program that calls the **$e** subcommand, and the output that results when the program runs (be sure to delete any previous breakpoints that you may have set):

```
.f+2:b,0$e
:r
adbsamp2: running
_errno: 0
```

```
_environ:  3fffe6bc
__NLinit:  10000238
_main: 100001ea
_exit: 1000028c
_fcnt: 0
_loop_count:  1
_f:   100001b4
_NLgetfile: 10000280
_write: 100002e0
__NLinit__X:  10000238
_NLgetfile__X:   10000280
__cleanup: 100002bc
__exit: 100002c8
_exit__X:  1000028c
__cleanup__X:  100002bc
breakpoint .f+2:  st r2,1c(r1)
```

Displaying the Address Maps

The **adb** program prepares a set of maps for the text and data segments in your program and uses these maps to access items that you request for display. Use the **$m** subcommand to display the contents of the address maps. For more information, see (“adb Debug Program Reference Information”). The subcommand displays the maps for all segments in the program and uses information taken from either the program and core files or directly from memory.

The **$m** subcommand displays information similar to the following:

```
$m
  [0] : ?map :   'adbsamp3'
  b1 = 0x10000200,  e1 = 0x10001839,  f1 = 0x10000200
  b2 = 0x2002c604,  e2 = 0x2002c8a4,  f2 = 0x600

  [1] : ?map :   'shr.o' in library 'lib/libc.a'
  b1 = 0xd00d6200,  e1 = 0xd013976f,  f1 = 0xd00defbc
  b2 = 0x20000600,  e2 = 0x2002bcb8,  f2 = 0x4a36c

  [-] :  /map :          '-'
  b1 = 0x0000000,  e1 = 0x00000000,  f1 = 0x00000000
  b2 = 0x0000000,  e2 = 0x00000000,  f2 = 0x00000000
```

The display defines address-mapping parameters for the text (`b1`, `e1`, and `f1`) and data (`b2`, `e2`, and `f2`) segments for the two files being used by the **adb** debug program. This example shows values for the **adbsamp3** sample program only. The second set of map values are for the core file being used. Since none was in use, the example shows the file name as **-** (dash).

The value displayed inside the square brackets can be used as the *Count* parameter in the **?e** and **?m** subcommands.

adb Debug Program Reference Information

The **adb** debug program uses addresses, expressions, operators, subcommands, and variables to organize and manipulate data.

adb Debug Program Addresses

The address in a file associated with a written address is determined by a mapping associated with that file. Each mapping is represented by two triples (*B1*, *E1*, *F1*) and (*B2*, *E2*, *F2*). The *FileAddress* parameter that corresponds to a written *Address* parameter is calculated as follows:

B1<=Address<E1=>FileAddress=Address+F1-B1

OR

B2<=Address<E2=>FileAddress=Address+F2-B2

If the requested *Address* parameter is neither between *B1* and *E1* nor between *B2* and *E2*, the *Address* parameter is not valid. In some cases, such as programs with separated I and D space, the two segments for a file may overlap. If a **?** (question mark) or **/** (slash) subcommand is followed by an ***** (asterisk), only the second triple is used.

The initial setting of both mappings is suitable for normal **a.out** and **core** files. If either file is not of the kind expected, the *B1* parameter for that file is set to a value of 0, the *E1* parameter is set to the maximum file size, and the *F1* parameter is set to a value of 0. In this way, the whole file can be examined with no address translation.

adb Debug Program Expressions

The following expressions are supported by the **adb** debug program:

. (period)	Specifies the last address used by a subcommand. The last address is also known as the current address.
+ (plus)	Increases the value of . (period) by the current increment.
^ (caret)	Decreases the value of . (period) by the current increment.
″ (double quotes)	Specifies the last address typed by a command.
Integer	Specifies an octal number if this parameter begins with **0o**, a hexadecimal number if preceded by **0x** or **#**, or a decimal number if preceded by **0t**. Otherwise, this expression specifies a number interpreted in the current radix. Initially, the radix is 16.
′*Cccc*’	Specifies the ASCII value of up to 4 characters. A \ (backslash) can be used to escape an ’ (apostrophe).
< *Name*	Reads the current value of the *Name* parameter. The *Name* parameter is either a variable name or a register name. The **adb** command maintains a number of variables named by single letters or digits. If the *Name* parameter is a register name, the value of the register is obtained from the system header in the *CoreFile* parameter. Use the **$r** subcommand to see the valid register names.
Symbol	Specifies a sequence of uppercase or lowercase letters, underscores, or digits, though the sequence cannot start with a digit. The value of the *Symbol* parameter is taken from the symbol table in the *ObjectFile* parameter. An initial _ (underscore) is prefixed to the *Symbol* parameter, if needed.
.*Symbol*	Specifies the entry point of the function named by the *Symbol* parameter.
Routine.Name	Specifies the address of the *Name* parameter in the specified C language routine. Both the *Routine* and *Name* parameters are *Symbol* parameters. If the *Name* parameter is omitted, the value is the address of the most recently activated C stack frame corresponding to the *Routine* parameter.
(*Expression***)**	Specifies the value of the expression.

adb Debug Program Operators

Integers, symbols, variables, and register names can be combined with the following operators:

Unary Operators

*****_Expression_	Returns contents of the location addressed by the *Expression* parameter in the *CoreFile* parameter.
@ *Expression*	Returns contents of the location addressed by the *Expression* parameter in the *ObjectFile* parameter.
-*Expression*	Performs integer negation.
~ *Expression*	Performs bit-wise complement.

Unary Operators

#*Expression*	Performs logical negation.

Binary Operators

*Expression1***+***Expression2*	Performs integer addition.
*Expression1***-***Expression2*	Performs integer subtraction.
*Expression1******Expression2*	Performs integer multiplication.
*Expression1***%***Expression2*	Performs integer division.
*Expression1***&***Expression2*	Performs bit-wise conjunction.
*Expression1***|***Expression2*	Performs bit-wise disjunction.
*Expression1***#***Expression2*	Rounds up the *Expression1* parameter to the next multiple of the *Expression2* parameter.

Binary operators are left-associative and are less binding than unary operators.

adb Debug Program Subcommands

You can display the contents of a text or data segment with the **?** (question mark) or the **/** (slash) subcommand. The **=** (equal sign) subcommand displays a given address in the specified format. The **?** and **/** subcommands can be followed by an ***** (asterisk).

?*Format*	Displays the contents of the *ObjectFile* parameter starting at the *Address* parameter. The value of **.** (period) increases by the sum of the increment for each format letter.
/*Format*	Displays the contents of the *CoreFile* parameter starting at the *Address* parameter. The value of **.** (period) increases by the sum of the increment for each format letter.
=*Format*	Displays the value of the *Address* parameter. The **i** and **s** format letters are not meaningful for this command.

The *Format* parameter consists of one or more characters that specify print style. Each format character may be preceded by a decimal integer that is a repeat count for the format character. While stepping through a format, the **.** (period) increments by the amount given for each format letter. If no format is given, the last format is used.

The available format letters are as follows:

a	Prints the value of **.** (period) in symbolic form. Symbols are checked to ensure that they have an appropriate type.
b	Prints the addressed byte in the current radix, unsigned.
c	Prints the addressed character.
C	Prints the addressed character using the following escape conventions: • Prints control characters as ~ (tilde) followed by the corresponding printing character. • Prints nonprintable characters as ~ (tilde) **<***Number***>**, where *Number* specifies the hexadecimal value of the character. The ~ character prints as ~ ~ (tilde tilde).
d	Prints in decimal.
D	Prints long decimal.
f	Prints the 32-bit value as a floating-point number.
F	Prints double floating point.
i *Number*	Prints as instructions. *Number* is the number of bytes occupied by the instruction.
n	Prints a new line.
o	Prints 2 bytes in octal.
O	Prints 4 bytes in octal.
p	Prints the addressed value in symbolic form using the same rules for symbol lookup as the **a** format letter.
q	Prints 2 bytes in the current radix, unsigned.

Q	Prints 4 unsigned bytes in the current radix.
r	Prints a space.
s *Number*	Prints the addressed character until a zero character is reached.
S *Number*	Prints a string using the ~ (tilde) escape convention. The *Number* variable specifies the length of the string including its zero terminator.
t	Tabs to the next appropriate tab stop when preceded by an integer. For example, the **8t** format command moves to the next 8-space tab stop.
u	Prints as an unsigned decimal number.
U	Prints a long unsigned decimal.
x	Prints 2 bytes in hexadecimal.
X	Prints 4 bytes in hexadecimal.
Y	Prints 4 bytes in date format.
/	Local or global data symbol.
?	Local or global text symbol.
=	Local or global absolute symbol.
"..."	Prints the enclosed string.
^	Decreases the **.** (period) by the current increment. Nothing prints.
+	Increases the **.** (period) by a value of 1. Nothing prints.
-	Decreases the **.** (period) decrements by a value of 1. Nothing prints.
newline	Repeats the previous command incremented with a *Count* of 1.
[**?/**]**l** *Value Mask*	Words starting at the **.** (period) are masked with the *Mask* value and compared with the *Value* parameter until a match is found. If **L** is used, the match is for 4 bytes at a time instead of 2 bytes. If no match is found, then **.** (period) is unchanged; otherwise, **.** (period) is set to the matched location. If the *Mask* parameter is omitted, a value of -1 is used.
[**?/**]**w** *Value***...**	Writes the 2-byte *Value* parameter into the addressed location. If the command is **W**, write 4 bytes. If the command is **V**, write 1 byte. Alignment restrictions may apply when using the **w** or **W** command.
[**,***Count*][**?/**]**m** *B1 E1 F1*[**?/**]	Records new values for the *B1*, *E1*, and *F1* parameters. If less than three expressions are given, the remaining map parameters are left unchanged. If the **?** (question mark) or **/** (slash) is followed by an ***** (asterisk), the second segment (*B2, E2, F2*) of the mapping is changed. If the list is terminated by **?** or **/**, the file (*ObjectFile* or *CoreFile*, respectively) is used for subsequent requests. (For example, the **/m?** command causes **/** to refer to the *ObjectFile*) file. If the *Count* parameter is specified, the **adb** command changes the maps associated with that file or library only. The **$m** command shows the count that corresponds to a particular file. If the *Count* parameter is not specified, a default value of 0 is used.
>*Name*	Assigns a . (period) to the variable or register specified by the *Name* parameter.
!	Calls a shell to read the line following **!** (exclamation mark).

$*Modifier* Miscellaneous commands. The available values for *Modifier* are:

<*File* Reads commands from the specified file and returns to standard input. If a count is given as 0, the command will be ignored. The value of the count is placed in the **adb 9** variable before the first command in the *File* parameter is executed.

<<*File* Reads commands from the specified file and returns to standard input. The **<<***File* command can be used in a file without causing the file to be closed. If a count is given as 0, the command is ignored. The value of the count is placed in the **adb 9** variable before the first command in *File* is executed. The **adb 9** variable is saved during the execution of the **<<***File* command and restored when **<<***File* completes. There is a limit to the number of **<<***File* commands that can be open at once.

>*File* Sends output to the specified file. If the *File* parameter is omitted, output returns to standard output. The *File* parameter is created if it does not exist.

b Prints all breakpoints and their associated counts and commands.

c Stacks back trace. If the *Address* parameter is given, it is taken as the address of the current frame (instead of using the frame pointer register). If the format letter **C** is used, the names and values of all automatic and static variables are printed for each active function. If the *Count* parameter is given, only the number of frames specified by the *Count* parameter are printed.

d Sets the current radix to the *Address* value or a value of 16 if no address is specified.

e Prints the names and values of external variables. If a count is specified, only the external variables associated with that file are printed.

f Prints the floating-point registers in hexadecimal.

i *instruction set*
Selects the instruction set to be used for disassembly.

l Changes the default directory as specified by the **-l** flag to the *Name* parameter value.

m Prints the address map.

n *mnem_set*
Selects the mnemonics to be used for disassembly.

o Sets the current radix to a value of 8.

q Exits the **adb** command.

r Prints the general registers and the instruction addressed by **iar** and sets the **.** (period) to **iar**. The *Number***$r** parameter prints the register specified by the *Number* variable. The *Number*,*Count***$r** parameter prints registers *Number*+*Count*-1,...,*Number*.

s Sets the limit for symbol matches to the *Address* value. The default is a value of 255.

v Prints all non-zero variables in octal.

w Sets the output page width for the *Address* parameter. The default is 80.

P *Name*
Uses the *Name* value as a prompt string.

? Prints the process ID, the signal that caused stoppage or termination, and the registers of **$r**.

:*Modifier* Manages a subprocess. Available modifiers are:

b*Command*
Sets the breakpoint at the *Address* parameter. The breakpoint runs the *Count* parameter -1 times before causing a stop. Each time the breakpoint is encountered, the specified command runs. If this command sets **.** (period) to a value of 0, the breakpoint causes a stop.

c*Signal* Continues the subprocess with the specified signal. If the *Address* parameter is given, the subprocess is continued at this address. If no signal is specified, the signal that caused the subprocess to stop is sent. Breakpoint skipping is the same as for the **r** modifier.

d Deletes the breakpoint at the *Address* parameter.

k Stops the current subprocess, if one is running.

r Runs the *ObjectFile* parameter as a subprocess. If the *Address* parameter is given explicitly, the program is entered at this point. Otherwise, the program is entered at its standard entry point. The *Count* parameter specifies how many breakpoints are to be ignored before stopping. Arguments to the subprocess can be supplied on the same line as the command. An argument starting with **<** or **>** establishes standard input or output for the command. On entry to the subprocess, all signals are turned on.

s*Signal* Continues the subprocess in single steps up to the number specified in the *Count* parameter. If there is no current subprocess, the *ObjectFile* parameter is run as a subprocess. In this case no signal can be sent. The remainder of the line is treated as arguments to the subprocess.

adb Debug Program Variables

The **adb** command provides a number of variables. When the **adb** program is started, the following variables are set from the system header in the specified core file. If the *CoreFile* parameter does not appear to be a **core** file, these values are set from the *ObjectFile* parameter:

0 Last value printed
1 Last displacement part of an instruction source
2 Previous value of the **1** variable
9 Count on the last **$<** or **$<<** subcommand
b Base address of the data segment
d Size of the data segment
e Entry address of the program
m ″Magic″ number
s Size of the stack segment
t Size of the text segment

Example adb Program: adbsamp

```
/* Program Listing for adbsamp.c */
char  str1[ ] = "This is a character string";
int one = 1;
int number = 456;
long lnum = 1234;
float fpt = 1.25;
char str2[ ] = "This is the second character string";
main()
{
        one = 2;
        printf("First String = %s\n",str1);
```

```
        printf("one = %d\n",one);
        printf("Number = %d\n",lnum);
        printf("Floating point Number = %g\n",fpt);
        printf("Second String = %s\n",str2);
}
```

Compile the program using the **cc** command to the **adbsamp** file as follows:

```
cc -g adbsamp.c -o adbsamp
```

To start the debug session, enter:

```
adb adbsamp
```

Example adb Program: adbsamp2

```
/*program listing for adbsamp2.c*/
int     fcnt,loop_count;
f(a,b)
int a,b;
{
        a = a+b;
        fcnt++;
        return(a);
}
main()
{
        loop_count = 0;
        while(loop_count <= 100)
        {
                loop_count = f(loop_count,1);
                printf("%s%d\n","Loop count is: ", loop_count);
                printf("%s%d\n","fcnt count is: ",fcnt);
        }
}
```

Compile the program using the **cc** command to the **adbsamp2** file with the following command:

```
cc -g adbsamp2.c -o adbsamp2
```

To start the debug session, enter:

```
adb adbsamp2
```

Example adb Program: adbsamp3

The following sample program **adbsamp3.c** contains an infinite recursion of subfunction calls. If you run this program to completion, it causes a memory fault error and quits.

```
int     fcnt,gcnt,hcnt;
h(x,y)
int x,y;
{
        int hi;
        register int hr;
        hi = x+1;
        hr = x-y+1;
        hcnt++;
        hj:
        f(hr,hi);
}
g(p,q)
int p,q;
{
        int gi;
        register int gr;
```

```
        gi = q-p;
        gr = q-p+1;
        gcnt++;
        gj:
        h(gr,gi);
}
f(a,b)
int a,b;
{
        int fi;
        register int fr;
        fi = a+2*b;
        fr = a+b;
        fcnt++;
        fj:
        g(fr,fi);
}
main()
{
        f(1,1);
}
```

Compile the program using the **cc** command to create the **adbsamp3** file with the following command:

```
cc -g adbsamp3.c -o adbsamp3
```

To start the debug session, enter:

```
adb adbsamp3
```

Example of Directory and i-node Dumps in adb Debugging

This example shows how to create **adb** scripts to display the contents of a directory and the i-node map of a file system. In the example, the directory is named **dir** and contains a variety of files. The file system is associated with the /dev/hd3 device file (**/tmp**), which has the necessary permissions to be read by the user.

To display a directory, create an appropriate script. A directory normally contains one or more entries. Each entry consists of an unsigned i-node number (i-number) and a 14-character file name. You can display this information by including a command in your script file. The **adb** debug program expects the object file to be an **xcoff** format file. This is not the case with a directory. The **adb** program indicates that the directory, because it is not an **xcoff** format file, has a text length of 0. Use the **m** command to indicate to the **adb** program that this directory has a text length of greater than 0. Therefore, display entries in your **adb** session by entering:

```
,0?m 360 0
```

For example, the following command displays the first 20 entries separating the i-node number and file name with a tab:

```
0,20?ut14cn
```

You can change the second number, `20`, to specify the number of entries in the directory. If you place the following command at the beginning of the script, the **adb** program displays the strings as headings for each column of numbers:

```
="inumber"8t"Name"
```

Once you have created the script file, redirect it as input when you start the **adb** program with the name of your directory. For example, the following command starts the **adb** program on the `geo` directory using command input from the `ddump` script file:

```
adb geo - <ddump
```

The minus sign (-) prevents the **adb** program from opening a core file. The **adb** program reads the commands from the script file.

To display the i-node table of a file system, create a new script and then start the **adb** program with the file name of the device associated with the file system. The i-node table of a file system has a complex structure. Each entry contains:

- A word value for status flags
- A byte value for number links
- 2-byte values for the user and group IDs
- A byte and word value for the size
- 8-word values for the location on disk of the file's blocks
- 2-word values for the creation and modification dates

The following is an example directory dump output:

```
        inumber Name
0:       26    .
        2     ..
        27    .estate
        28    adbsamp
        29    adbsamp.c
        30    calc.lex
        31    calc.yacc
        32    cbtest
        68    .profile
        66    .profile.bak
        46    adbsamp2.c
        52    adbsamp2
        35    adbsamp.s
        34    adbsamp2.s
        48    forktst1.c
        49    forktst2.c
        50    forktst3.c
        51    lpp&us1.name
        33    adbsamp3.c
        241   sample
        198   adbsamp3
        55    msgqtst.c
        56    newsig.c
```

The i-node table starts at the address 02000. You can display the first entry by putting the following command in your script file:

```
02000,-1?on3bnbrdn8un2Y2na
```

The command specifies several new-line characters for the output display to make it easier to read.

To use the script file with the i-node table of the **/dev/hd3** file, enter the following command:

```
adb /dev/hd3 - <script
```

Each entry in the display has the form:

```
02000: 073145
    0163 0164 0141
    0162 10356
    28770 8236 25956 27766 25455 8236 25956 25206
    1976 Feb 5 08:34:56 1975 Dec 28 10:55:15
```

Example of Data Formatting in adb Debugging

To display the current address after each machine instruction, enter:

```
main , 5 ? ia
```

This produces output such as the following when used with the example program **adbsamp**:

```
.main:
.main:          mflr 0
.main+4:        st r0,  0x8(r1)
.main+8:        stu rs,    (r1)
.main+c:        li l  r4,  0x1
.main+10:       oril  r3, r4, 0x0
.main+14:
```

To make it clearer that the current address does not belong to the instruction that appears on the same line, add the new-line format character (n) to the command:

```
.main , 5 ? ian
```

In addition, you can put a number before a formatting character to indicate the number of times to repeat that format.

To print a listing of instructions and include addresses after every fourth instruction, use the following command:

```
.main,3?4ian
```

This instruction produces the following output when used with the example program **adbsamp**:

```
.main:
                mflr 0
                st r0, 0x8(r1)
                stu r1, -56(r1)
                lil r4, 0x1

.main+10:
                oril r3, r4, 0x0
                bl .f
                l r0, 0x40(r1)
                ai r1, r1, 0x38

.main+20:
                mtlr r0
                br
                Invalid opcode
                Invalid opcode

.main+30:
```

Be careful where you put the number.

The following command, though similar to the previous command, does not produce the same output:

```
main,3?i4an
.main:
.main:          mflr  0
.main+4:         .main+4:        .main+4:         .main+4:
                 st  r0,  0x8(r1)
.main+8:        .main+8:         .main+8:         .main+8:
                stu  r1,    (r1)
.main+c:        .main+c:         .main+c:         .main+c:
```

You can combine format requests to provide elaborate displays. For example, entering the following command displays instruction mnemonics followed by their hexadecimal equivalent:

```
.main,-1?i^xn
```

In this example, the display starts at the address `main`. The negative count (-1) causes an indefinite call of the command, so that the display continues until an error condition (such as end-of-file) occurs. In the format, `i` displays the mnemonic instruction at that location, the `^` (caret) moves the current address back to the beginning of the instruction, and `x` re-displays the instruction as a hexadecimal number. Finally, `n` sends a newline character to the terminal. The output is similar to the following, only longer:

```
.main:
.main:          mflr  0
                7c0802a6
                  st r0, 0x8(r1)
                9001008
                  st r1, -56(r1)
                9421ffc8
                  lil r4, 0x1
                38800001
                  oril r3, r4, 0x0
                60830000
                  bl  -  .f
                4bffff71
                  l r0, 0x40(r1)
                80010040
                  ai r1, r1, 0x38
                30210038
                  mtlr r0
                7c0803a6
```

The following example shows how to combine formats in the **?** or **/** subcommand to display different types of values when stored together in the same program. It uses the **adbsamp** program. For the commands to have variables with which to work, you must first set a breakpoint to stop the program, and then run the program until it finds the breakpoint. Use the **:b** command to set a breakpoint:

```
.main+4:b
```

Use the **$b** command to show that the breakpoint is set:

```
$b
breakpoints
count  bkpt      command
1    .main+4
```

Run the program until it finds the breakpoint by entering:

```
:r
adbsamp: running
breakpoint   .main+4:    st r0, 0x8(r1)
```

You can now display conditions of the program when it stopped. To display the value of each individual variable, give its name and corresponding format in a **/** (slash) command. For example, the following command displays the contents of `str1` as a string:

```
str1/s
str1:
str1:     This is a character string
```

The following command displays the contents of `number` as a decimal integer:

```
number/D
number:
number:    456
```

You can choose to view a variable in a variety of formats. For example, you can display the long variable **lnum** as a 4-byte decimal, octal, and hexadecimal number by entering the commands:

```
lnum/D
lnum:
lnum:     1234

lnum/O
lnum:
lnum:     2322

lnum/X
lnum:
lnum:     4d2
```

You can also examine variables in other formats. For example, the following command displays some variables as eight hexadecimal values on a line and continues for five lines:

```
str1,5/8x
str1:
str1:   5468  6973  2069  7320  6120  6368  6172  6163
         7465  7220  7374  7269  6e67  0   0   0     0

number:  0        1c8   0     0      0    4d2    0     0
         3fa0    0   0     0    5468  6973  2069  7320
         7468  6520  7365  636f  6e64  2063  6861  7261
```

Since the data contains a combination of numeric and string values, display each value as both a number and a character to see where the actual strings are located. You can do this with one command:

```
str1,5/4x4^8Cn
str1:
str1:    5468 6973   2069  7320  This is
          6120  6368  6172  6163   a charac
          7465  7220  7374  7269   ter stri
          6e67  0      0    0       ng~@~@~@~@~@
          0    1c8   0       0      ~@~@~A~<c8>~@~@~@~@
```

In this case, the command displays four values in hexadecimal, then displays the same values as eight ASCII characters. The ^ (caret) is used four times just before displaying the characters to set the current address back to the starting address for that line.

To make the display easier to read, you can insert a tab between the values and characters and give an address for each line:

```
str1,5/4x4^8t8Cna
str1:
str1:     5468  6973  2069  7320      This is
str1+8:   6120  6368  6172  6163      a charac
str1+10:  7465  7220  7374  7269      ter stri
str1+18:  6e67     0    0    1        ng~@~@~@~@~@~A

number:
number:    0      1c8   0     0       ~@~@~A~<c8>~@~@~@~@
fpt:
```

Example of Tracing Multiple Functions in adb Debugging

> **Note:** The example program used in this section, **adbsamp3**, contains an infinite recursion of subfunction calls. If you run this program to completion, it causes a memory fault error and quits.

The following example shows how to execute a program under **adb** control and carry out the basic debugging operations described in the following sections.

The source program for this example is stored in a file named **adbsamp3.c**. Compile this program to an executable file named **adbsamp3** using the **cc** command:

```
cc adbsamp3.c -o adbsamp3
```

Starting the adb Program

To start the session and open the program file, use the following command (no core file is used):

```
adb adbsamp3
```

Setting Breakpoints

First, set breakpoints at the beginning of each function using the **:b** subcommand:

```
.f:b
.g:b
.h:b
```

Displaying a Set of Instructions

Next, display the first five instructions in the **f** function:

```
.f,5?ia
.f:
.f:             mflr  r0
.f+4:           st  r0, 0x8(r1)
.f+8:           stu  r1, -64(r1)
.f+c:           st  r3,  0x58(r1)
.f+10:          st  r4,  0x5c(r1)
.f+14:
```

Display five instructions in function **g** without their addresses:

```
.g,5?i
.g:    mflr  r0
        st  r0, 0x8(r1)
        stu  r1, -64(r1)
        st  r3,  0x58(r1)
        st  r4,  0x5c(r1)
```

Starting the adsamp3 Program

Start the program by entering the following command:

```
:r
adbsamp3: running
breakpoint   .f:       mflr  r0
```

The **adb** program runs the sample program until it reaches the first breakpoint where it stops.

Removing a Breakpoint

Since running the program to this point causes no errors, you can remove the first breakpoint:

```
.f:d
```

Continuing the Program

Use the **:c** subcommand to continue the program:

```
:c
adbsamp3: running
breakpoint   .g:       mflr  r0
```

The **adb** program restarts the **adbsamp3** program at the next instruction. The program operation continues until the next breakpoint, where it stops.

Tracing the Path of Execution

Trace the path of execution by entering:

```
$c
.g(0,0) .f+2a
.f(1,1) .main+e
.main(0,0,0) start+fa
```

The **$c** subcommand displays a report that shows the three active functions: `main`, `f` and `g`.

Displaying a Variable Value

Display the contents of the `fcnt` integer variable by entering the command:

```
fcnt/D
fcnt:
fcnt:     1
```

Skipping Breakpoints

Next, continue running the program and skip the first 10 breakpoints by entering:

```
,10:c
adbsamp3: running
breakpoint   .g:       mflr  r0
```

The **adb** program starts the **adbsamp3** program and displays the running message again. It does not stop the program until exactly 10 breakpoints have been encountered. To ensure that these breakpoints have been skipped, display the backtrace again:

```
$c
.g(0,0) .f+2a
.f(2,11) .h+28
.h(10,f) .g+2a
.g(11,20) .f+2a
.f(2,f) .h+28
.h(e,d) .g+2a
.g(f,1c) .f+2a
.f(2,d) .h+28
.h(c,b) .g+2a
.g(d,18) .f+2a
.f(2,b) .h+28
.h(a,9) .g+2a
.g(b,14) .f+2a
.f(2,9) .h+28
.h(8,7) .g+2a
.g(9,10) .f+2a
.f(2,7) .h+28
.h(6,5) .g+2a
.g(7,c) .f+2ae
.f(2,5) .h+28
.h(4,3) .g+2a
.g(5,8) .f+2a
.f(2,3) .h+28
.h(2,1) .g+2a
.g(2,3) .f+2a
.f(1,1) .main+e
.main(0,0,0) start+fa
```

dbx Symbolic Debug Program Overview

The **dbx** symbolic debug program allows you to debug a program at two levels: the source-level and the assembler language-level. Source level debugging allows you to debug your C, C++, Pascal, or FORTRAN language program. Assembler language level debugging allows you to debug executable programs at the machine level. The commands used for machine level debugging are similar to those used for source-level debugging.

Using the **dbx** debug program, you can step through the program you want to debug one line at a time or set breakpoints in the object program that will stop the debug program. You can also search through and display portions of the source files for a program.

The following sections contain information on how to perform a variety of tasks with the **dbx** debug program:

- "Using the dbx Debug Program"
- "Displaying and Manipulating the Source File with the dbx debug Program" on page 66
- "Examining Program Data" on page 70
- "Debugging at the Machine Level with dbx" on page 76
- "Customizing the dbx Debugging Environment" on page 78

Using the dbx Debug Program

The following sections contain information on how to use the **dbx** debug program.

Starting the dbx Debug Program

The **dbx** program can be started with a variety of flags. The three most common ways to start a debug session with the **dbx** program are:

- Running the **dbx** command on a specified object file
- Using the **-r** flag to run the **dbx** command on a program that ends abnormally
- Using the **-a** flag to run the **dbx** command on a process that is already in progress

When the **dbx** command is started, it checks for a **.dbxinit** ("Using the .dbxinit File" on page 79) file in the user's current directory and in the user's **$HOME** directory. If a **.dbxinit** file exists, its subcommands run at the beginning of the debug session. If a **.dbxinit** file exists in both the home and current directories, then both are read in that order. Because the current directory **.dbxinit** file is read last, its subcommands can supercede those in the home directory.

If no object file is specified, then the **dbx** program asks for the name of the object file to be examined. The default is **a.out**. If the **core** file exists in the current directory or a *CoreFile* parameter is specified, then the **dbx** program reports the location where the program faulted. Variables, registers, and memory held in the core image may be examined until execution of the object file begins. At that point the **dbx** debug program prompts for commands.

Running Shell Commands from dbx

You can run shell commands without exiting from the debug program using the **sh** subcommand.

If **sh** is entered without any commands specified, the shell is entered for use until it is exited, at which time control returns to the **dbx** program.

Command Line Editing in dbx

The **dbx** command provides command line editing features similar to those provided by **Korn Shell**. **vi** mode provides **vi**-like editing features, while **emacs** mode gives you controls similar to **emacs**.

You can turn these features on by using **dbx** subcommand **set -o** or **set edit**. So, to turn on **vi**-style command line editing, you would type the subcommand `set edit vi` or `set -o vi`.

You can also use the **EDITOR** environment variable to set the editing mode.

The **dbx** command saves commands entered to history file **.dbxhistory**. If the **DBXHISTFILE** environment variable is not set, then the history file used is **$HOME/.dbxhistory**.

By default, the **dbx** command saves the text of the last 128 commands entered. The **DBXHISTSIZE** environment variable can be used to increase this limit.

Using Program Control

The **dbx** debug program allows you to set breakpoints (stopping places) in the program. After entering the **dbx** program you can specify which lines or addresses are to be breakpoints and then run the program you want to debug with the **dbx** program. The program halts and reports when it reaches a breakpoint. You can then use **dbx** commands to examine the state of your program.

An alternative to setting breakpoints is to run your program one line or instruction at a time, a procedure known as single-stepping.

Setting and Deleting Breakpoints

Use the **stop** subcommand to set breakpoints in the **dbx** program. The **stop** subcommand halts the application program when certain conditions are fulfilled:

- The *Variable* is changed when the *Variable* parameter is specified.
- The *Condition* is true when the **if** *Condition* flag is used.
- The *Procedure* is called when the **in** *Procedure* flag is used.
- The *SourceLine* line number is reached when the **at** *SourceLine* flag is used.

> **Note:** The *SourceLine* variable can be specified as an integer or as a file name string followed by a : (colon) and an integer.

After any of these commands, the **dbx** program responds with a message reporting the event ID associated with your breakpoint along with an interpretation of your command.

Running a Program

The **run** subcommand starts your program. It tells the **dbx** program to begin running the object file, reading any arguments just as if they were typed on the shell command line. The **rerun** subcommand has the same form as **run**; the difference is that if no arguments are passed, the argument list from the previous execution is used. After your program begins, it continues until one of the following events occurs:

- The program reaches a breakpoint.
- A signal occurs that is not ignored, such as **INTERRUPT** or **QUIT**.
- A multiprocess event occurs while multiprocess debugging is enabled.
- The program performs a **load**, **unload**, or **loadbind** subroutine.

Note: The **dbx** program ignores this condition if the **$ignoreload** debug variable is set. This is the default. For more information see the **set** subcommand.

- The program completes.

In each case, the **dbx** debug program receives control and displays a message explaining why the program stopped.

There are several ways to continue the program once it stops:

cont	Continues the program from where it stopped.
detach	Continues the program from where it stopped, exiting the debug program. This is useful after you have patched the program and want to continue without the debug program.
return	Continues execution until a return to *Procedure* is encountered, or until the current procedure returns if *Procedure* is not specified.
skip	Continues execution until the end of the program or until *Number* + 1 breakpoints execute.
step	Runs one or a specified *Number* of source lines.
next	Runs up to the next source line, or runs a specified *Number* of source lines.

A common method of debugging is to step through your program one line at a time. The **step** and **next** subcommands serve that purpose. The distinction between these two commands is apparent only when the next source line to be run involves a call to a subprogram. In this case, the **step** subcommand stops in the subprogram; the **next** subcommand runs until the subprogram has finished and then stops at the next instruction after the call.

The **$stepignore** debug variable can be used to modify the behavior of the **step** subcommand. See the **dbx** command in *AIX 5L Version 5.1 Commands Reference, Volume 2* for more information.

There is no event number associated with these stops because there is no permanent event associated with stopping a program.

If your program has multiple threads, they all run normally during the **cont**, **next**, **nexti**, and **step** subcommands. These commands act on the running thread (the thread which stopped execution by hitting a breakpoint), so even if another thread executes the code which is being stepped, the **cont**, **next**, **nexti**, or **step** operation continues until the running thread has also executed that code.

If you want these subcommands to execute the running thread only, you can set the **dbx** debug program variable **$hold_next**; this causes the **dbx** debug program to hold all other user threads during **cont**, **next**, **nexti**, and **step** subcommands.

Note: If you use this feature, remember that a held thread will not be able to release any locks which it has acquired; another thread which requires one of these locks could deadlock your program.

Separating dbx Output from Program Output

Use the **screen** subcommand for debugging programs that are screen-oriented, such as text editors or graphics programs. This subcommand opens an Xwindow for **dbx** command interaction. The program continues to operate in the window in which it originated. If **screen** is not used, **dbx** program output is intermixed with the screen-oriented program output.

Tracing Execution

The **trace** subcommand tells the **dbx** program to print information about the state of the program being debugged while that program is running. The **trace** subcommand can slow a program considerably, depending on how much work the **dbx** program has to do. There are five forms of program tracing:

- You can single-step the program, printing out each source line that is executed. The **$stepignore** debug variable can be used to modify the behavior of the **trace** subcommand. See the **set** subcommand for more information.
- You can restrict the printing of source lines to when the specified procedure is active. You can also specify an optional condition to control when trace information is produced.
- You can display a message each time a procedure is called or returned.
- You can print the specified source line when the program reaches that line.
- You can print the value of an expression when the program reaches the specified source line.

Deleting trace events is the same as deleting stop events. When the **trace** subcommand is executed, the event ID associated is displayed along with the internal representation of the event.

Displaying and Manipulating the Source File with the dbx debug Program

You can use the **dbx** debug program to search through and display portions of the source files for a program.

You do not need a current source listing for the search. The **dbx** debug program keeps track of the current file, current procedure, and current line. If a core file exists, the current line and current file are set initially to the line and file containing the source statement where the process ended.

Note: This is only true if the process stopped in a location compiled for debugging.

- "Debugging Programs Involving Multiple Threads" on page 67
- "Displaying and Modifying Variables" on page 72

Changing the Source Directory Path

By default, the **dbx** debug program searches for the source file of the program being debugged in the following directories:

- Directory where the source file was located when it was compiled. This directory is searched only if the compiler placed the source path in the object.
- Current directory.
- Directory where the program is currently located.

You can change the list of directories to be searched by using the **-I** option on the **dbx** invocation line or issuing the **use** subcommand within the **dbx** program. For example, if you moved the source file to a new location since compilation time, you might want to use one of these commands to specify the old location, the new location, and some temporary location.

Displaying the Current File

The **list** subcommand allows you to list source lines.

The $ (dollar sign) and @ (at sign) symbols represent *SourceLineExpression* and are useful with the **list**, **stop**, and **trace** subcommands. The $ symbol represents the next line to be run. The @ symbol represents the next line to be listed.

The **move** subcommand changes the next line number to be listed.

Changing the Current File or Procedure

Use the **func** and **file** subcommands to change the current file, current procedure, and current line within the **dbx** program without having to run any part of your program.

Search through the current file for text that matches regular expressions. If a match is found, the current line is set to the line containing the matching text. The syntax of the search subcommand is:

/ *RegularExpression* **[/]**	Searches forward in the current source file for the given expression.
? *RegularExpression* **[?]**	Searches backward in the current source file for the given expression.

If you repeat the search without arguments, the **dbx** command searches again for the previous regular expression. The search wraps around the end or beginning of the file.

You can also invoke an external text editor for your source file using the **edit** subcommand. You can override the default editor (**vi**) by setting the **EDITOR** environment variable to your desired editor before starting the **dbx** program.

The **dbx** program resumes control of the process when the editing session is completed.

Debugging Programs Involving Multiple Threads

Programs involving multiple user threads call the subroutine **pthread_create**. When a process calls this subroutine, the operating system creates a new thread of execution within the process. When debugging a multi-threaded program, it is necessary to work with individual threads instead of with processes. The **dbx** program only works with user threads: in the **dbx** documentation, the word *thread* is usually used alone to mean *user thread*. The **dbx** program assigns a unique thread number to each thread in the process being debugged, and also supports the concept of a running and current thread:

Running thread	The user thread that was responsible for stopping the program by hitting a breakpoint. Subcommands that single-step the program work with the running thread.
Current thread	The user thread that you are examining. Subcommands that display information work in the context of the current thread.

By default, the running thread and current thread are the same. You can select a different current thread by using the **thread** subcommand. When the **thread** subcommand displays threads, the current thread line is preceded by a **>**. If the running thread is not the same as the current thread, its line is preceded by a *****.

Identifying Thread-Related Objects

Threads use mutexes and condition variables to synchronize access to resources. Threads, mutexes, and condition variables are created with attribute objects that define how they behave. The **dbx** program automatically creates several variables that identify these various thread-related objects. For each object class, **dbx** maintains a numbered list and creates an associated variable for each object in the list. These variable names begin with a $ (dollar sign), followed by a letter indicating the object class (**a**, **c**, **m**, or **t**), followed by a number indicating the object's position in the class list. The letters and their associated object classes are as follows:

- **a** for attributes
- **c** for condition variables
- **m** for mutexes
- **t** for threads.

For example, **$t2** corresponds to the second thread in the **dbx** thread list. In this case, **2** is the object's thread number, which is unrelated to the kernel thread identifier (tid). You can list the objects in each class using the following **dbx** subcommands: **attribute**, **condition**, **mutex**, and **thread**. For example, you can simply use the **thread** subcommand to list all threads.

The **dbx** program automatically defines and maintains the variable **$running_thread**, which identifies the thread that was running when a breakpoint was hit.

Breakpoints and Threads

If your program has multiple user threads, simply setting a breakpoint on a source line will not guarantee that a particular thread will hit the breakpoint, because several threads can execute the same code. If any thread hits the breakpoint, all the threads of the process will stop.

If you want to specify which thread is to hit the breakpoint, you can use the **stop** or **stopi** subcommands to set a conditional breakpoint. The following aliases set the necessary conditions automatically:

- **bfth** (*Function*, *ThreadNumber*)
- **blth** (*LineNumber*, *ThreadNumber*)

These aliases stop the thread at the specified function or source line number, respectively. *ThreadNumber* is the number part of the symbolic thread name as reported by the **thread** subcommand (for example, 2 is the *ThreadNumber* for the thread name $t2).

For example, the following subcommand stops thread `$t1` at function `func1`:

```
(dbx) bfth (func1, 1)
```

and the following subcommand stops thread `$t2` at source line 103:

```
(dbx) blth (103, 2)
```

If no particular thread was specified with the breakpoint, any thread that executes the code where the breakpoint is set could become the running thread.

Thread-Related subcommands

The **dbx** debug program has the following subcommands that enable you to work with individual attribute objects, condition variables, mutexes, and threads:

attribute	Displays information about all attribute objects, or attribute objects specified by attribute number.
condition	Displays information about all condition variables, condition variables that have waiting threads, condition variables that have no waiting threads, or condition variables specified by condition number.
mutex	Displays information about all mutexes, locked or unlocked mutexes, or mutexes specified by mutex number.
thread	Displays information about threads, selects the current thread, and holds and releases threads.

A number of subcommands that do not deal with threads directly are also affected when used to debug a multi-threaded program:

print

If passed a symbolic object name reported by the **thread**, **mutex**, **condition**, or **attribute** subcommands, displays status information about the object. For example, to display the third mutex and the first thread:

```
(dbx) print $m3, $t1
```

stop, stopi
If a single thread hits a breakpoint, all other threads are stopped as well, and the process timer is halted. This means that the breakpoint does not affect the global behavior of the process. These normal breakpoints are global, meaning that they can stop any thread.

If you want to specify which thread will hit the breakpoint, you must use a condition as shown in the following example, which ensures that only thread `$t5` can hit the breakpoint set on function `f1`:

```
(dbx) stopi at &f1 if ($running_thread == 5)
```

This syntax also works with the **stop** subcommand. Another way to specify these conditions is to use the **bfth** and **blth** aliases, as explained in the section ″Breakpoints and Threads″ (“Breakpoints and Threads” on page 68).

step, **next, nexti**
All threads resume execution during the **step, next,** and **nexti** subcommands. If you want to step the running thread only, set the **$hold_next dbx** debug program variable; this holds all threads except the running thread during these subcommands.

stepi
The **stepi** subcommand executes the specified number of machine instructions in the running thread only. Other threads in the process being debugged will not run during the **stepi** subcommand.

trace, tracei
A specific user thread can be traced by specifying a condition with the **trace** and **tracei** subcommands as shown in the following example, which traces changes made to `var1` by thread `$t1`:

```
(dbx) trace var1 if ($running_thread == 1)
```

If a multi-threaded program does not protect its variables with mutexes, the **dbx** debug program behavior may be affected by the resulting race conditions. For example, suppose that your program contains the following lines:

```
59 var = 5;

60 printf(″var=%d\n″, var);
```

If you want to verify that the variable is being initialized correctly, you could type:

```
stop at 60 if var==5
```

The **dbx** debug program puts a breakpoint at line 60, but if access to the variable is not controlled by a mutex, another thread could update the variable before the breakpoint is hit. This means that the **dbx** debug program would not see the value of five and would continue execution.

Debugging Programs Involving Multiple Processes

Programs involving multiple processes call the **fork** and **exec** subroutines. When a program forks, the operating system creates another process that has the same image as the original. The original process is called the parent process, the created process is called the child process.

When a process performs an **exec** subroutine, a new program takes over the original process. Under normal circumstances, the debug program debugs only the parent process. However, the **dbx** program can follow the execution and debug the new processes when you issue the **multproc** subcommand. The **multproc** subcommand enables multiprocess debugging.

When multiprocess debugging is enabled and a fork occurs, the parent and child processes are halted. A separate virtual terminal Xwindow is opened for a new version of the **dbx** program to control running of the child process:

```
(dbx) multproc on
(dbx) multproc
multi-process debugging is enabled
(dbx) run
```

When the fork occurs, execution is stopped in the parent, and the **dbx** program displays the state of the program:

```
application forked, child pid = 422, process stopped, awaiting input
stopped due to fork with multiprocessing enabled in fork at 0x1000025a (fork+0xe)
(dbx)
```

Another virtual terminal Xwindow is then opened to debug the child process:

```
debugging child, pid=422, process stopped, awaiting input
stopped due to fork with multiprocessing enabled in fork at 0x10000250
10000250 (fork+0x4) )80010010    1       r0,0x10(r1)
(dbx)
```

At this point, two distinct debugging sessions are running. The debugging session for the child process retains all the breakpoints from the parent process, but only the parent process can be rerun.

When a program performs an **exec** subroutine in multiprocess debugging mode, the program overwrites itself, and the original symbol information becomes obsolete. All breakpoints are deleted when the **exec** subroutine runs; the new program is stopped and identified for the debugging to be meaningful. The **dbx** program attaches itself to the new program image, makes a subroutine to determine the name of the new program, reports the name, and then prompts for input. The prompt is similar to the following:

```
(dbx) multproc
Multi-process debugging is enabled
(dbx) run
Attaching to program from exec . . .
Determining program name . . .
Successfully attached to /home/user/execprog . . .
Reading symbolic information . . .
(dbx)
```

If a multi-threaded program forks, the new child process will have only one thread. The process should call the **exec** subroutine. Otherwise, the original symbol information is retained, and thread-related subcommands (such as **thread**) display the objects of the parent process, which are obsolete. If an **exec** subroutine is called, the original symbol information is reinitialized, and the thread-related subcommands display the objects in the new child process.

It is possible to follow the child process of a fork without a new Xwindow being opened by using the **child** flag of the **multproc** subcommand. When a forked process is created, **dbx** follows the child process. The **parent** flag of the **multproc** subcommand causes **dbx** to stop when a program forks, but then follows the parent. Both the **child** and **parent** flags follow an execed process. These flags are very useful for debugging programs when Xwindows is not running.

Examining Program Data

This section explains how to examine, test, and modify program data.

Handling Signals

The **dbx** debug program can either trap or ignore signals before they are sent to your program. Each time your program is to receive a signal, the **dbx** program is notified. If the signal is to be ignored, it is passed to your program; otherwise, the **dbx** program stops the program and notifies you that a signal has been trapped. The **dbx** program cannot ignore the **SIGTRAP** signal if it comes from a process outside of the debug process. In a multi-threaded program, a signal can be sent to a particular thread via the

pthread_kill subroutine. By default, the **dbx** program stops and notifies you that a signal has been trapped. If you request a signal be passed on to your program using the **ignore** subcommand, the **dbx** program ignores the signal and passes it on to the thread. Use the **catch** and **ignore** subcommands to change the default handling.

In the following example, a program uses **SIGGRANT** and **SIGREQUEST** to handle allocation of resources. In order for the **dbx** program to continue each time one of these signals is received, enter:

```
(dbx) ignore GRANT
(dbx) ignore SIGREQUEST
(dbx) ignore
CONT CLD ALARM KILL GRANT REQUEST
```

The **dbx** debug program can block signals to your program if you set the **$sigblock** variable. By default, signals received through the **dbx** program are sent to the source program or the object file specified by the **dbx** *ObjectFile* parameter. If the **$sigblock** variable is set using the **set** subcommand, signals received by the **dbx** program are not passed to the source program. If you want a signal to be sent to the program, use the **cont** subcommand and supply the signal as an operand.

You can use this feature to interrupt execution of a program running under the **dbx** debug program. Program status can be examined before continuing execution as usual. If the **$sigblock** variable is not set, interrupting execution causes a **SIGINT** signal to be sent to the program. This causes execution, when continued, to branch to a signal handler if one exists.

The following example program illustrates how execution using the **dbx** debug program changes when the **$sigblock** variable is set:

```
#include <signal.h>
#include <stdio.h>
void inthand( ) {
        printf("\nSIGINT received\n");
        exit(0);
}
main( )
{
        signal(SIGINT, inthand);
        while (1) {
                  printf(".");
                  fflush(stdout);
                sleep(1);
        }
}
```

The following sample session with the **dbx** program uses the preceding program as the source file. In the first run of the program, the **$sigblock** variable is not set. During rerun, the **$sigblock** variable is set. Comments are placed between angle brackets to the right:

```
dbx version 3.1.
Type 'help' for help.
reading symbolic information ...
(dbx) run
.........^C                      <User pressed Ctrl-C here!>
interrupt in sleep at 0xd00180bc
0xd00180bc (sleep+0x40) 80410014         1        r2,0x14(r1)
(dbx) cont
SIGINT received
execution completed
(dbx) set $sigblock
(dbx) rerun
[ looper ]
..............^C                  <User pressed Ctrl-C here!>
interrupt in sleep at 0xd00180bc
```

```
0xd00180bc (sleep+0x40) 80410014          1        r2,0x14(r1)
(dbx) cont
....^C   <Program did not receive signal, execution continued>
interrupt in sleep at 0xd00180bc
0xd00180bc (sleep+0x40) 80410014          1         r2,0x14(r1)
(dbx) cont 2                     <End program with a signal 2>
SIGINT received
execution completed
(dbx)
```

Calling Procedures

You can call your program procedures from the **dbx** program to test different arguments. You can also call diagnostic routines that format data to aid in debugging. Use the **call** subcommand or the **print** subcommand to call a procedure.

Displaying a Stack Trace

To list the procedure calls preceding a program halt, use the **where** command.

In the following example, the executable object file, **hello**, consists of two source files and three procedures, including the standard procedure `main`. The program stopped at a breakpoint in procedure `sub2`.

```
(dbx) run
[1] stopped in sub2 at line 4 in file "hellosub.c"
(dbx) where
sub2(s = "hello", n = 52), line 4 in "hellosub.c"
sub(s = "hello", a = -1, k = delete), line 31 in "hello.c"
main(), line 19 in "hello.c"
```

The stack trace shows the calls in reverse order. Starting at the bottom, the following events occurred:

1. Shell called `main`.
2. `main` called `sub` procedure at line 19 with values `s = "hello"`, `a = -1`, and `k = delete`.
3. `sub` called `sub2` procedure at line 31 with values `s = "hello"` and `n = 52`.
4. The program stopped in `sub2` procedure at line 4.

> **Note:** Set the debug program variable **$noargs** to turn off the display of arguments passed to procedures.

You can also display portions of the stack with the **up** and **down** subcommands.

Displaying and Modifying Variables

To display an expression, use the **print** subcommand. To print the names and values of variables, use the **dump** subcommand. If the given procedure is a period, then all active variables are printed. To modify the value of a variable, use the **assign** subcommand.

In the following example, a C program has an automatic integer variable `x` with value 7, and `s` and `n` parameters in the `sub2` procedure:

```
(dbx) print x, n
7 52
(dbx) assign x = 3*x
(dbx) print x
```

```
21
(dbx) dump
sub2(s = "hello", n = 52)
x = 21
```

Displaying Thread-Related Information

To display information on user threads, mutexes, conditions, and attribute objects, use the **thread**, **mutex**, **condition**, and **attribute** subcommands. You can also use the **print** subcommand on these objects. In the following example, the running thread is thread 1. The user sets the current thread to be thread 2, lists the threads, prints information on thread 1, and finally prints information on several thread-related objects.

```
(dbx) thread current 2
(dbx) thread
 thread  state-k   wchan state-u   k-tid mode held scope function
*$t1     run             running   12755   u   no   pro  main
>$t2     run             running   12501   k   no   sys  thread_1
```

```
(dbx) print $t1
(thread_id = 0x1, state = run, state_u = 0x0, tid = 0x31d3, mode = 0x1, held = 0x0, priority = 0x3c,
    policy = other, scount = 0x1, cursig = 0x5, attributes = 0x200050f8)
```

```
(dbx) print $a1,$c1,$m2
(attr_id = 0x1, type = 0x1, state = 0x1, stacksize = 0x0, detachedstate = 0x0, process_shared = 0x0,
 contentionscope = 0x0, priority = 0x0, sched = 0x0, inherit = 0x0, protocol = 0x0, prio_ceiling = 0x0)
(cv_id = 0x1, lock = 0x0, semaphore_queue = 0x200032a0, attributes = 0x20003628)
(mutex_id = 0x2, islock = 0x0, owner = (nil), flags = 0x1, attributes = 0x200035c8)
```

Scoping of Names

Names resolve first using the static scope of the current function. The dynamic scope is used if the name is not defined in the first scope. If static and dynamic searches do not yield a result, an arbitrary symbol is chosen and the message `using QualifiedName` is printed. You can override the name resolution procedure by qualifying an identifier with a block name (such as *Module.Variable*). Source files are treated as modules named by the file name without the suffix. For example, the *x* variable, which is declared in the `sub` procedure inside the **hello.c** file, has the fully qualified name **hello.sub.x**. The program itself has a period for a name.

The **which** and **whereis** subcommands can be helpful in determining which symbol is found when multiple symbols with the same name exist.

Using Operators and Modifiers in Expressions

The **dbx** program can display a wide range of expressions. Specify expressions with a common subset of C and Pascal syntax, with some FORTRAN extensions.

***** (asterisk) or ^ (caret)	Denotes indirection or pointer dereferencing.
[ ] (brackets) or **( )** (parentheses)	Denotes subscript array expressions.
. (period)	Use this field reference operator with pointers and structures. This makes the C operator -> (arrow) unnecessary, although it is allowed.
& (ampersand)	Gets the address of a variable.
.. (two periods)	Separates the upper and lower bounds when specifying a subsection of an array. For example: **n[1..4]**.

The following types of operations are valid in expressions:

Algebraic	**=**, **-**, *****,**/**(floating division), **div** (integral division), **mod**, **exp** (exponentiation)
Bitwise	**-**, **I**, **bitand**, **xor**, ~, **<<**, **>>**

Logical	**or**, **and**, **not**, **||**, **&&**
Comparison	**<**, **>**, **<=**, **>=**, **<>** or **!=**, **=** or **==**
Other	**sizeof**

Logical and comparison expressions are allowed as conditions in **stop** and **trace** subcommands.

Checking of Expression Types

The **dbx** debug program checks expression types. You can override the expression type by using a renaming or casting operator. There are three forms of type renaming:

- *Typename* **(***Expression***)**
- *Expression* **** *Typename*
- **(***Typename***)** *Expression*

> **Note:** When you cast to or from a structure, union, or class, the casting is left-justified. However, when casting from a class to a base class, C++ syntax rules are followed.

For example, to rename the x variable where x is an integer with a value of 97, enter:

```
(dbx) print char (x), x \ char, (char) x, x,
'a' 'a' 'a' 97
```

The following examples show how you can use the **(***Typename***)** *Expression* form of type renaming:

```
print (float) i
print ((struct qq *) void_pointer)->first_element
```

The following restrictions apply to C-style typecasting for the **dbx** debug program:

- The FORTRAN types (integer*1, integer*2, integer*4, logical*1, logical*2, logical*4, and so on) are not supported as cast operators.
- If an active variable has the same name as one of the base types or user-defined types, the type cannot be used as a cast operator for C-style typecasting.

The **whatis** subcommand prints the declaration of an identifier, which you can then qualify with block names.

Use the **$$***TagName* construct to print the declaration of an enumeration, structure, or union tag (or the equivalent in Pascal).

The type of the **assign** subcommand expression must match the variable type you assigned. If the types do not match, an error message is displayed. Change the expression type using a type renaming. Disable type checking by setting a special **dbx** debug program **$unsafeassign** variable.

Folding Variables to Lowercase and Uppercase

By default, the **dbx** program folds symbols based on the current language. If the current language is C, C++, or undefined, the symbols are not folded. If the current language is FORTRAN or Pascal, the symbols are folded to lowercase. The current language is undefined if the program is in a section of code that has not been compiled with the **debug** flag. You can override default handling with the **case** subcommand.

Using the **case** subcommand without arguments displays the current case mode.

The FORTRAN and Pascal compilers convert all program symbols to lowercase; the C compiler does not.

Changing Print Output with Special Debug Program Variables

Use the **set** subcommand to set the following special **dbx** debug program variables to get different results from the **print** subcommand:

$hexints	Prints integer expressions in hexadecimal.
$hexchars	Prints character expressions in hexadecimal.
$hexstrings	Prints the address of the character string, not the string itself.
$octints	Prints integer expressions in octal.
$expandunions	Prints fields within a union.
$pretty	Displays complex C and C++ types in **pretty** format.

Set and unset the debug program variables to get the desired results. For example:

```
(dbx) whatis x; whatis i; whatis s
int x;
char i;
char *s;
(dbx) print x, i, s
375 'c' "hello"
(dbx) set $hexstrings; set $hexints; set $hexchars
(dbx) print x, i, s
0x177 0x63 0x3fffe460
(dbx) unset $hexchars; set $octints
(dbx) print x, i
0567 'c'
(dbx) whatis p
struct info p;
(dbx) whatis struct info
struct info {
    int x;
    double position[3];
    unsigned char c;
    struct vector force;
};
(dbx) whatis struct vector
struct vector {
    int a;
    int b;
    int c;
};
(dbx) print p
(x = 4, position = (1.3262493258532527e-315, 0.0, 0.0), c = '\0', force = (a = 0, b = 9, c = 1))
(dbx) set $pretty="on"
(dbx) print p
{
    x = 4
    position[0] = 1.3262493258532527e-315
    position[1] = 0.0
    position[2] = 0.0
    c = '\0'
    force = {
        a = 0
        b = 9
        c = 1
    }
}
(dbx) set $pretty="verbose"
(dbx) print p
x = 4
position[0] = 1.3262493258532527e-315
position[1] = 0.0
position[2] = 0.0
```

```
c = '\0'
force.a = 0
force.b = 9
force.c = 1
```

Debugging at the Machine Level with dbx

You can use the **dbx** debug program to examine programs at the assembly language level. You can display and modify memory addresses, display assembler instructions, single-step instructions, set breakpoints and trace events at memory addresses, and display the registers.

In the commands and examples that follow, an address is an expression that evaluates to a memory address. The most common forms of addresses are integers and expressions that take the address of an identifier with the **&** (ampersand) operator. You can also specify an address as an expression enclosed in parentheses in machine-level commands. Addresses can be composed of other addresses and the operators **+** (plus), **-** (minus), and indirection (unary *****).

Using Machine Registers

Use the **registers** subcommand to see the values of the machine registers. Registers are divided into three groups: general-purpose, floating-point, and system-control.

General-purpose registers

General-purpose registers are denoted by **$r***Number*, where *Number* represents the number of the register.

> **Note:** The register value may be set to a hexadecimal value of `0xdeadbeef`. This is an initialization value assigned to all general-purpose registers at process initialization.

Floating-point registers

Floating-point registers are denoted by **$fr***Number*, where *Number* represents the number of the register. Floating-point registers are not displayed by default. Unset the **$noflregs** debug program variable to enable the floating-point register display (unset **$noflregs**).

System-control registers

Supported system-control registers are denoted by:

- The Instruction Address register, **$iar** or **$pc**
- The Condition Status register, **$cr**
- The Multiplier Quotient register, **$mq**
- The Machine State register, **$msr**
- The Link register, **$link**
- The Count register, **$ctr**
- The Fixed Point Exception register, **$xer**
- The Transaction ID register, **$tid**
- The Floating-Point Status register, **$fpscr**

Examining Memory Addresses

Use the following command format to print the contents of memory starting at the first address and continuing up to the second address, or until the number of items specified by the *Count* variable are displayed. The *Mode* specifies how memory is to print.

*Address***,** *Address* **/ [***Mode***][>** *File***]**

Address **/ [***Count***][***Mode***] [>** *File***]**

If the *Mode* variable is omitted, the previous mode specified is reused. The initial mode is **X**. The following modes are supported:

b Prints a byte in octal.
c Prints a byte as a character.
D Prints a long word in decimal.
d Prints a short word in decimal.
f Prints a single-precision floating-point number.
g Prints a double-precision floating-point number.
h Prints a byte in hexadecimal.
i Prints the machine instruction.
lld Prints an 8-byte signed decimal number.
llo Prints an 8-byte unsigned octal number.
llu Prints an 8-byte unsigned decimal number.
llx Prints an 8-byte unsigned hexadecimal number.
O Prints a long word in octal.
o Prints a short word in octal.
q Prints an extended-precision floating-point number.
s Prints a string of characters terminated by a null byte.
X Prints a long word in hexadecimal.
x Prints a short word in hexadecimal.

In the following example, expressions in parentheses can be used as an address:

```
(dbx) print &x
0x3fffe460
(dbx) &x/X
3fffe460: 31323300
(dbx) &x,&x+12/x
3fffe460: 3132 3300 7879 7a5a 5958 5756 003d 0032
(dbx) ($pc)/2i
100002cc (sub)  7c0802a6        mflr     r0
100002d0 (sub + 0x4)    bfc1fff8         stm       r30,-8(r1)
```

Running a Program at the Machine Level

The commands for debugging your program at the machine-level are similar to those at the symbolic level. The **stopi** subcommand stops the machine when the address is reached, the condition is true, or the variable is changed. The **tracei** subcommands are similar to the symbolic trace commands. The **stepi** subcommand executes either one or the specified *Number* of machine instructions.

If you performed another **stepi** subcommand at this point, you would stop at address 0x10000618, identified as the entry point of procedure `printf`. If you do not intend to stop at this address, you could use the **return** subcommand to continue execution at the next instruction in `sub` at address 0x100002e0. At this point, the **nexti** subcommand will automatically continue execution to 0x10000428.

If your program has multiple threads, the symbolic thread name of the running thread is displayed when the program stops. For example:

```
stopped in sub at 0x100002d4 ($t4)
10000424 (sub+0x4) 480001f5 bl 0x10000618 (printf)
```

Debugging fdpr Reordered Executables

You can debug programs that have been reordered with **fdpr** (feedback directed program restructuring, part of Performance Toolbox for AIX) at the instruction level. If optimization options **-R0** or **-R2** are used, additional information is provided enabling **dbx** to map most reordered instruction addresses to the corresponding addresses in the original executable as follows:

```
0xRRRRRRRR = fdpr[0xYYYYYYYY]
```

In this example, `0xRRRRRRRR` is the reordered address and `0xYYYYYYYY` is the original address. In addition, **dbx** uses the traceback entries in the original instruction area to find associated procedure names for the `stopped in` message, the **func** subcommand, and the traceback.

```
(dbx) stepi
stopped in proc_d at 0x1000061c = fdpr[0x10000278]
0x1000061c (???) 9421ffc0       stwu   r1,-64(r1)
(dbx)
```

In the preceding example, **dbx** indicates the program is stopped in the `proc_d` subroutine at address `0x1000061c` in the reordered text section originally located at address `0x10000278`. For more information about **fdpr**, see the **fdpr** command.

Displaying Assembly Instructions

The **listi** subcommand for the **dbx** command displays a specified set of instructions from the source file. In the default mode, the **dbx** program lists the instructions for the architecture on which it is running. You can override the default mode with the **$instructionset** and **$mnemonics** variables of the **set** subcommand for the **dbx** command.

For more information on displaying instructions or disassembling instructions, see the **listi** subcommand for the **dbx** command. For more information on overriding the default mode, see the **$instructionset** and **$mnemonics** variables of the **set** subcommand for the **dbx** command.

Customizing the dbx Debugging Environment

You can customize the debugging environment by creating subcommand aliases and by specifying options in the **.dbxinit** file. You can read **dbx** subcommands from a file using the **-c** flag. The following sections contain more information about customization options.

Defining a New dbx Prompt

The **dbx** prompt is normally the name used to start the **dbx** program. If you specified `/usr/ucb/dbx a.out` on the command line, then the prompt is `/usr/ucb/dbx`.

You can change the prompt with the **prompt** subcommand, or by specifying a different prompt in the **prompt** line of the **.dbxinit** file. Changing the prompt in the **.dbxinit** file causes your prompt to be used instead of the default each time you initialize the **dbx** program.

For example, to initialize the **dbx** program with the debug prompt `debug–>`, enter the following line in your **.dbxinit** file:

```
prompt "debug-->"
```

Creating dbx Subcommand Aliases

You can build your own commands from the **dbx** primitive subcommand set. The following commands allow you to build a user alias from the arguments specified. All commands in the replacement string for the alias must be **dbx** primitive subcommands. You can then use your aliases in place of the **dbx** primitives.

The **alias** subcommand with no arguments displays the current aliases in effect; with one argument the command displays the replacement string associated with that alias.

alias [*AliasName*[*CommandName*]]

alias *AliasName "CommandString"*

alias *AliasName* **(***Parameter1, Parameter2, . . .* **)** *"CommandString"*

The first two forms of the **alias** subcommand are used to substitute the replacement string for the **alias** each time it is used. The third form of aliasing is a limited macro facility. Each parameter specified in the **alias** subcommand is substituted in the replacement string.

The following aliases and associated subcommand names are defaults:

attr	attribute
bfth	**stop** (in given thread at specified function)
blth	**stop** (in given thread at specified source line)
c	cont
cv	condition
d	delete
e	edit
h	help
j	status
l	list
m	map
mu	mutex
n	next
p	**print**
q	quit
r	run
s	step
st	stop
t	**where**
th	thread
x	registers

You can remove an alias with the **unalias** command.

Using the .dbxinit File

Each time you begin a debugging session, the **dbx** program searches for special initialization files named **.dbxinit**, which contain lists of **dbx** subcommands to execute. These subcommands are executed before the **dbx** program begins to read subcommands from standard input. When the **dbx** command is started, it checks for a **.dbxinit** file in the user's current directory and in the user's **$HOME** directory. If a **.dbxinit** file exists, its subcommands run at the beginning of the debug session. If a **.dbxinit** file exists in both the home and current directories, then both are read in that order. Because the current directory **.dbxinit** file is read last, its subcommands can supercede those in the home directory.

Normally, the **.dbxinit** file contains **alias** subcommands, but it can contain any valid **dbx** subcommands. For example:

```
$ cat .dbxinit
alias si "stop in"
prompt "dbg-->"
$ dbx a.out
dbx version 3.1
Type 'help' for help.
reading symbolic information . . .
dbg--> alias
si   stop in
t    where . . .
dbg-->
```

Reading dbx Subcommands from a File

The **-c** invocation option and **.dbxinit** file provide mechanisms for executing **dbx** subcommands before reading from standard input. When the **-c** option is specified, the **dbx** program does not search for a **.dbxinit** file. Use the **source** subcommand to read **dbx** subcommands from a file once the debugging session has begun.

After executing the list of commands in the **cmdfile** file, the **dbx** program displays a prompt and waits for input.

You can also use the **-c** option to specify a list of subcommands to be executed when initially starting the **dbx** program.

List of dbx Subcommands

The commands and subcommands for the **dbx** debug program are located in the *AIX 5L Version 5.1 Commands Reference*.

The **dbx** debug program provides subcommands for performing the following task categories:

Setting and Deleting Breakpoints

clear	Removes all stops at a given source line.
cleari	Removes all breakpoints at an address.
delete	Removes the traces and stops corresponding to the specified numbers.
status	Displays the currently active **trace** and **stop** subcommands.
stop	Stops execution of the application program.

Running Your Program

cont	Continues running the program from the current breakpoint until the program finishes or another breakpoint is encountered.
detach	Exits the debug program, but continues running the application.
down	Moves a function down the stack.
goto	Causes the specified source line to be the next line run.
gotoi	Changes program counter addresses.
next	Runs the application program up to the next source line.
nexti	Runs the application program up to the next source instruction.
rerun	Begins running an application.

return Continues running the application program until a return to the specified procedure is reached.
run Begins running an application.
skip Continues execution from the current stopping point.
step Runs one source line.
stepi Runs one source instruction.
up Move a function up the stack.

Tracing Program Execution

trace Prints tracing information.
tracei Turns on tracing.
where Displays a list of all active procedures and functions.

Ending Program Execution

quit Quits the **dbx** debug program.

Displaying the Source File

edit Invokes an editor on the specified file.
file Changes the current source file to the specified file.
func Changes the current function to the specified function or procedure.
list Displays lines of the current source file.
listi Lists instructions from the application.
move Changes the next line to be displayed.
/ (Search) Searches forward in the current source file for a pattern.
? (Search) Searches backward in the current source file for a pattern.
use Sets the list of directories to be searched when looking for a file.

Printing and Modifying Variables, Expressions, and Types

assign Assigns a value to a variable.
case Changes the way in which **dbx** interprets symbols.
dump Displays the names and values of variables in the specified procedure.
print Prints the value of an expression or runs a procedure and prints the return code.
set Assigns a value to a nonprogram variable.
unset Deletes a nonprogram variable.
whatis Displays the declaration of application program components.
whereis Displays the full qualifications of all the symbols whose names match the specified identifier.
which Displays the full qualification of the specified identifier.

Thread Debugging

attribute Displays information about all or selected attributes objects.
condition Displays information about all or selected condition variables.
mutex Displays information about all or selected mutexes.
thread Displays and controls threads.

Multiprocess Debugging

multproc	Enables or disables multiprocess debugging.

Procedure Calling

call	Runs the object code associated with the named procedure or function.
print	Prints the value of an expression or runs a procedure and prints the return code.

Signal Handling

catch	Starts trapping a signal before that signal is sent to the application program.
ignore	Stops trapping a signal before that signal is sent to the application program.

Machine-Level Debugging

display memory	Displays the contents of memory.
gotoi	Changes program counter addresses.
map	Displays address maps and loader information for the application program.
nexti	Runs the application program up to the next machine instruction.
registers	Displays the values of all general-purpose registers, system-control registers, floating-point registers, and the current instruction register.
stepi	Runs one source instruction.
stopi	Sets a stop at a specified location.
tracei	Turns on tracing.

Debugging Environment Control

alias	Displays and assigns aliases for **dbx** subcommands.
help	Displays help information for **dbx** subcommands or topics.
prompt	Changes the **dbx** prompt to the specified string.
screen	Opens an Xwindow for **dbx** command output.
sh	Passes a command to the shell for execution.
source	Reads **dbx** commands from a file.
unalias	Removes an alias.

Chapter 4. Error Notification

Each time an error is logged, the **error notification** daemon determines if the error log entry matches the selection criteria of any of the Error Notification objects. If matches exist, the daemon runs the programmed action, also called a notify method, for each matched object.

The Error Notification object class is located in the **/etc/objrepos/errnotify** file. Error Notification objects are added to the object class by using Object Data Manager (ODM) commands. Error Notification objects contain the following descriptors:

en_alertflg
Identifies whether the error is alertable. This descriptor is provided for use by alert agents associated with network management applications. The valid alert descriptor values are:

TRUE alertable

FALSE not alertable

en_class
Identifies the class of the error log entries to match. The valid **en_class** descriptor values are:

H Hardware Error class

S Software Error class

O Messages from the **errlogger** command

U Undetermined

en_crcid
Specifies the error identifier associated with a particular error.

en_label
Specifies the label associated with a particular error identifier as defined in the output of the **errpt -t** command.

en_method
Specifies a user-programmable action, such as a shell script or command string, to be run when an error matching the selection criteria of this Error Notification object is logged. The error notification daemon uses the **sh -c** command to execute the notify method.

The following key words are automatically expanded by the **error notification** daemon as arguments to the notify method.

$1 Sequence number from the error log entry

$2 Error ID from the error log entry

$3 Class from the error log entry

$4 Type from the error log entry

$5 Alert flags value from the error log entry

$6 Resource name from the error log entry

$7 Resource type from the error log entry

$8 Resource class from the error log entry

$9 Error label from the error log entry

en_name
Uniquely identifies the object. The creator uses this unique name when removing the object.

en_persistenceflg Designates whether the Error Notification object should be automatically removed when the system is restarted. For example, to avoid erroneous signaling, Error Notification objects containing methods which send a signal to another process should not persist across system restarts. This is because the receiving process and its process ID do not persist across system restarts.

The creator of the Error Notification object is responsible for removing the Error Notification object at the appropriate time. In the event that the process terminates and fails to remove the Error Notification object, the **en_persistenceflg** descriptor ensures that obsolete Error Notification objects are removed when the system is restarted.

The valid **en_persistenceflg** descriptor values are:

0 non-persistent (removed at boot time)

1 persistent (persists through boot)

en_pid Specifies a process ID (PID) for use in identifying the Error Notification object. Objects that have a PID specified should have the **en_persistenceflg** descriptor set to 0.

en_rclass Identifies the class of the failing resource. For the hardware error class, the resource class is the device class. The resource error class is not applicable for the software error class.

en_resource Identifies the name of the failing resource. For the hardware error class, a resource name is the device name.

en_rtype Identifies the type of the failing resource. For the hardware error class, a resource type is the device type a resource is known by in the devices object class.

en_symptom Enables notification of an error accompanied by a symptom string when set to **TRUE**.

en_type Identifies the severity of error log entries to match. The valid **en_type** descriptor values are:

INFO Informational

PEND Impending loss of availability

PERM Permanent

PERF Unacceptable performance degradation

TEMP Temporary

UNKN Unknown

TRUE Matches alertable errors.

FALSE Matches non-alertable errors.

0 Removes the Error Notification object at system restart.

non-zero
Retains the Error Notification object at system restart.

Security

Only processes running with the root user authority can add objects to the Error Notification object class.

Examples

1. To create a notify method that mails a formatted error entry to root each time a disk error of type PERM is logged, create a file called `/tmp/en_sample.add` containing the following Error Notification object:

```
errnotify:
    en_name = "sample"
    en_persistenceflg = 0
    en_class = "H"
    en_type = "PERM"
    en_rclass = "disk"
    en_method = "errpt -a -l $1 | mail -s 'Disk Error' root"
```

To add the object to the Error Notification object class, enter:

```
odmadd /tmp/en_sample.add
```

The **odmadd** command adds the Error Notification object contained in `/tmp/en_sample.add` to the `errnotify` file.

2. To verify that the Error Notification object was added to the object class, enter:

```
odmget -q"en_name='sample'" errnotify
```

The **odmget** command locates the Error Notification object within the `errnotify` file that has an **en_name** value of `"sample"` and displays the object. The following output is returned:

```
errnotify:
    en_pid = 0
    en_name = "sample"
    en_persistenceflg = 0
    en_label = ""
    en_crcid = 0
    en_class = "H"
    en_type = "PERM"
    en_alertflg = ""
    en_resource = ""
    en_rtype = ""
    en_rclass = "disk"
    en_method = "errpt -a -l $1 | mail -s 'Disk Error' root"
```

3. To delete the `sample` Error Notification object from the Error Notification object class, enter:

```
odmdelete -q"en_name='sample'" -o errnotify
```

The **odmdelete** command locates the Error Notification object within the `errnotify` file that has an **en_name** value of `"sample"` and removes it from the Error Notification object class.

Related Information

Error Logging Special Files in *AIX 5L Version 5.1 Files Reference*.

The **errdemon** daemon in *AIX 5L Version 5.1 Commands Reference*.

The **errclear** command, **errdead** command, **errinstall** command, **errlogger** command, **errmsg** command, **errpt** command, **errstop** command, **errupdate** command, **odmadd** command, **odmdelete** command, **odmget** command in *AIX 5L Version 5.1 Commands Reference*.

The **errlog** subroutine in *AIX 5L Version 5.1 Technical Reference: Base Operating System and Extensions Volume 1*.

The **errsave** kernel service in *AIX 5L Version 5.1 Technical Reference: Kernel and Subsystems Volume 1*.

Error Logging Facility

The error logging facility records hardware and software failures in the error log for information purposes or for fault detection and corrective action.

Refer to the following to use the error logging facility:

- "Error Logging Overview"
- "Managing Error Logging" on page 87
- "Error Logging Tasks" on page 90
- "Error Logging and Alerts" on page 97
- "Error Logging Controls" on page 98

In AIX Version 4 some of the error log commands are delivered in an optionally installable package called **bos.sysmgt.serv_aid**. The base system (**bos.rte**) includes the services for logging errors to the error log file. This includes the errlog subroutines, the **errsave** and **errlast** kernel service, the error device driver (**/dev/error**), the **error** daemon, and the **errstop** command. The commands required for licensed program installation (**errinstall** and **errupdate**) are also included in **bos.rte**. See "for information on Software Service Aids Package." Also, for information on transferring your system's error log file to a system that has the Software Service Aids package installed, see "Transferring Your Error Log to Another System" on page 87.

Error Logging Overview

The error logging process begins when an operating system module detects an error. The error-detecting segment of code then sends error information to either the **errsave** and **errlast** kernel service or the **errlog** application subroutine, where the information is in turn written to the **/dev/error** special file. This process then adds a time stamp to the collected data. The **errdemon** daemon constantly checks the **/dev/error** file for new entries, and when new data is written, the daemon conducts a series of operations.

Before an entry is written to the error log, the **errdemon** daemon compares the label sent by the kernel or application code to the contents of the Error Record Template Repository. If the label matches an item in the repository, the daemon collects additional data from other parts of the system.

To create an entry in the error log, the **errdemon** daemon retrieves the appropriate template from the repository, the resource name of the unit that detected the error and detail data. Also, if the error signifies a hardware-related problem and hardware vital product data (VPD) exists, the daemon retrieves the VPD from the Object Data Manager. When you access the error log, either through SMIT or with the **errpt** command, the error log is formatted according to the error template in the error template repository and presented in either a summary or detailed report. Most entries in the error log are attributable to hardware and software problems, but informational messages can also be logged.

The **diag** command uses the error log in part to diagnose hardware problems. To correctly diagnose new system problems, the system deletes hardware-related entries older than 90 days from the error log. The system deletes software-related entries 30 days after they are logged.

Terms to help you use the error logging facility include the following:

error ID
A 32-bit CRC hexadecimal code used to identify a particular failure. Each error record template has a unique error ID.

error label
The mnemonic name for an error ID.

error log
The file that stores instances of errors and failures encountered by the system.

error log entry	A record in the system error log that describes a hardware failure, a software failure, or an operator message. An error log entry contains captured failure data.
error record template	A description of what will be displayed when the error log is formatted for a report, including information on the type and class of the error, probable causes, and recommended actions. Collectively, the templates comprise the Error Record Template Repository.

Managing Error Logging

Error logging is automatically started during system initialization by the **rc.boot** script and is automatically stopped during system shutdown by the **shutdown** script. The error log analysis performed by the diagnostics (**diag** command) analyzes hardware error entries up to 90 days old. If you remove hardware error entries less than 90 days old, you can limit the effectiveness of this error log analysis.

To manage error logging efficiently, see:

- "Transferring Your Error Log to Another System"
- "Configuring Error Logging"
- "Removing Error Log Entries" on page 89
- "Enabling and Disabling Logging for an Event" on page 89
- "Setting Up Error Notification" on page 90
- "Logging Maintenance Activities" on page 90

Transferring Your Error Log to Another System

The **errclear**, **errdead**, **errlogger**, **errmsg**, and **errpt** commands are part of the optionally installable Software Service Aids package (**bos.sysmgt.serv_aid**). You need the Software Service Aids package to generate reports from the error log or delete entries from the error log. You can install the Software Service Aids package on your system or you can transfer your system's error log file to a system that has the Software Service Aids package installed.

Determine the path to your system's error log file by running the following command:

```
/usr/lib/errdemon -l
```

There are a number of ways to transfer the file to another system. For example, you can copy the file to a remotely mounted file system using the **cp** command; you can copy the file across the network connection using the **rcp**, **ftp**, or **tftp** commands; or you can copy the file to removable media using the **tar** or **backup** command and restore the file onto another system.

You can format reports for an error log copied to your system from another system by using the **-i** flag of the **errpt** command. The **-i** flag allows you to specify the path name of an error log file other than the default. Likewise, you can delete entries from an error log file copied to your system from another system by using the **-i** flag of the **errclear** command.

Configuring Error Logging

You can customize the name and location of the error log file and the size of the internal error buffer to suit your needs.

You can also control the logging of duplicate errors.

Listing the Current Settings
To list the current settings, run **/usr/lib/errdemon -l**. The values for the error log file name, error log file size, and buffer size that are currently stored in the error log configuration database display on your screen.

To list the current settings, run **/usr/lib/errdemon -l**. The values for the error log file name, error log file size, buffer size, and duplicate handling values that are currently stored in the error log configuration database display on your screen.

Customizing the Log File Location
To change the filename used for error logging run the **/usr/lib/errdemon -i** *FileName* command. The specified file name is saved in the error log configuration database and the error daemon is immediately restarted.

Customizing the Log File Size
To change the maximum size of the error log file enter:

```
/usr/lib/errdemon -s LogSize
```

The specified log file size limit is saved in the error log configuration database and the error daemon is immediately restarted. If the log file size limit is smaller than the size of the log file currently in use, the current log file is renamed by appending **.old** to the file name and a new log file is created with the specified size limit. The amount of space specified is reserved for the error log file and is not available for use by other files. Therefore, you should be careful not to make the log excessively large. But, if you make the log too small, important information may be overwritten prematurely. When the log file size limit is reached, the file *wraps*, that is, the oldest entries are overwritten by new entries.

Customizing the Buffer Size
To change the size of the error log device driver's internal buffer, enter:

```
/usr/lib/errdemon -B BufferSize
```

The specified buffer size is saved in the error log configuration database and, if it is larger than the buffer size currently in use, the in-memory buffer is immediately increased. If it is smaller than the buffer size currently in use, the new size is put into effect the next time the error daemon is started after the system is rebooted. The buffer cannot be made smaller than the hard-coded default of 8KB. The size you specify is rounded up to the next integral multiple of the memory page size (4KBs). The memory used for the error log device driver's in-memory buffer is not available for use by other processes (the buffer is pinned).

You should be careful not to impact your system's performance by making the buffer excessively large. But, if you make the buffer too small, the buffer may become full if error entries are arriving faster than they are being read from the buffer and put into the log file. When the buffer is full, new entries are discarded until space becomes available in the buffer. When this situation occurs, an error log entry is created to inform you of the problem, and you should correct the problem by enlarging the buffer.

Customizing Duplicate Error Handling
By default, starting with AIX 5.1, the error daemon eliminates duplicate errors. It does this by looking at each error that is logged. An error is a duplicate if it is identical to the previous error, and occurs within the approximate time interval specified with **/usr/lib/errdemon -t time-interval**. The default time value is 100, .1 seconds. The value is in milliseconds.

The **-m maxdups** flag controls how many duplicates can build up before a duplicate entry is logged. The default value is 1000. If an error, followed by 1000 occurrences of the same error, is logged, a duplicate error will be logged at that point rather than waiting for the time interval to expire or a unique error.

Thus if, for example, a device handler starts logging many identical errors rapidly, most will not appear in the log. Rather, the first occurrence will be logged as it is today. Subsequent occurrences will not be

logged immediately, just counted. When the time interval expires, the **maxdups** value is reached, or when another error is logged, an alternate form of the error is logged giving the times of the first and last duplicate and how many duplicates there were.

> **Note:** The time interval refers to the time since the last error, not the time since the first occurrence of this error, (i.e.) it is reset each time an error is logged. Also note that to be a duplicate, an error must exactly match the previous error. If, for example, anything about the detail data is different from the previous error, then that error is considered unique and logged as a separate error.

Removing Error Log Entries

Entries are removed from the error log when the root user runs the **errclear** command, when the **errclear** command is automatically invoked by a daily **cron** job, and when the error log file wraps as a result of reaching its maximum size. When the error log file reaches the maximum size specified in the error log configuration database, the oldest entries are overwritten by the newest entries.

Automatic Removal

The system is shipped with a **crontab** file to delete hardware errors older than 90 days and other errors older than 30 days. To display the **crontab** entries for your system, enter:

```
crontab -l Command
```

To change these entries, enter:

```
crontab -e Command
```

See the **crontab** command.

errclear Command

The **errclear** command can be used to selectively remove entries from the error log. The selection criteria you may specify include the error id number, sequence number, error label, resource name, resource class, error class, and error type. You must also specify the age of entries to be removed. The entries that match the selection criteria you specified and are older than the number of days you specified will be removed.

Enabling and Disabling Logging for an Event

You can disable logging or reporting of a particular event by modifying the Log or the Report field of the error template for the event. You can use the **errupdate** command to change the current settings for an event.

Showing Events for Which Logging is Disabled

To list all events for which logging is currently disabled, enter:

```
errpt -t -F Log=0
```

Events for which logging is disabled are not saved in the error log file.

Showing Events for which Reporting is Disabled

To list all events for which reporting is currently disabled, enter:

```
errpt -t -F Report=0
```

Events for which reporting is disabled are saved in the error log file when they occur, but they are not displayed by the **errpt** command.

Changing the Current Setting for an Event

You can use the **errupdate** command to change the current settings for an event. The necessary input to the **errupdate** command can be in a file or from standard input.

In the following example, standard input is used. To disable the reporting of the **ERRLOG_OFF** event (error id number 192AC071), enter the following lines to run the **errupdate** command:

```
errupdate <Enter>
=192AC071: <Enter>
Report=False <Enter>
<Ctrl-D>
<Ctrl-D>
```

Setting Up Error Notification

Refer to “Chapter 4. Error Notification” on page 83 in *AIX 5L Version 5.1 General Programming Concepts: Writing and Debugging Programs*.

Logging Maintenance Activities

The **errlogger** command allows the system administrator to record messages in the error log. Whenever you perform a maintenance activity, such as clearing entries from the error log, replacing hardware, or applying a software fix, it is a good idea to record this activity in the system error log.

Redirecting syslog Messages to Error Log

Some applications use syslog for logging errors and other events. Some administrators find it desirable to be able to list error log messages and syslog messages in a single report. This can be accomplished by redirecting the syslog messages to the error log. You can do this by specifying *errlog* as the destination in the syslog configuration file **(/etc/syslog.conf**). See the **syslogd** daemon for more information.

Directing Error Log Messages to Syslog

You can log error log events in the **syslog** file by using the **logger** command with the concurrent error notification capabilities of error log. For example, to log system messages (syslog), add an errnotify object with the following contents:

```
errnotify:
        en_name = "syslog1"
        en_persistenceflg = 1
        en_method = "logger Msg from Error Log: 'errpt -l $1 | grep -v 'ERROR_ID TIMESTAMP''"
```

For example, create a file called **/tmp/syslog.add** with these contents, then run the command **odmadd /tmp/syslog.add** (you must be logged in as root to do this).

For more information about concurrent error notification, see the “Chapter 4. Error Notification” on page 83 in *AIX 5L Version 5.1 General Programming Concepts: Writing and Debugging Programs*.

Error Logging Tasks

Error logging tasks and information to assist you in using the error logging facility include:

- “Reading an Error Report”
- “Examples of Detailed Error Reports” on page 92
- “Example of a Summary Error Report” on page 95
- “Generating an Error Report” on page 95
- “Stopping an Error Log” on page 96
- “Cleaning an Error Log” on page 96
- “Copying an Error Log to Diskette or Tape” on page 97

Reading an Error Report

To obtain a report of all errors logged in the 24 hours prior to the failure, enter:

```
errpt -a -s mmddhhmmyy | pg
```

where *mmddhhmmyy* represents the month, day, hour, minute, and year 24 hours prior to the failure.

An error log report contains the following information:

Note: Not all errors will generate information for each of the following categories.

LABEL Predefined name for the event.

ID Numerical identifier for the event.

Date/Time Date and time of the event.

Sequence Number Unique number for the event.

Machine ID Identification number of your system processor unit.

Node ID Mnemonic name of your system.

Class General source of the error. The possible error classes are:

- **H** Hardware. (When you receive a hardware error, refer to your system operator guide for information about performing diagnostics on the problem device or other piece of equipment. The diagnostics program tests the device and analyze the error log entries related to it to determine the state of the device.)
- **S** Software.
- **O** Informational messages.
- **U** Undetermined (for example, a network).

Type Severity of the error that has occurred. Five types of errors are possible:

- **PEND** The loss of availability of a device or component is imminent.
- **PERF** The performance of the device or component has degraded to below an acceptable level.
- **PERM** Condition that could not be recovered from. Error types with this value are usually the most severe errors and are more likely to mean that you have a defective hardware device or software module. Error types other than `PERM` usually do not indicate a defect, but they are recorded so that they can be analyzed by the diagnostics programs.
- **TEMP** Condition that was recovered from after a number of unsuccessful attempts. This error type is also used to record informational entries, such as data transfer statistics for DASD devices.
- **UNKN** It is not possible to determine the severity of the error.
- **INFO** The error log entry is informational and was not the result of an error.

Resource Name Name of the resource that has detected the error. For software errors. this is the name of a software component or an executable program. For hardware errors, this is the name of a device or system component. It does not indicate that the component is faulty or needs replacement. Instead, it is used to determine the appropriate diagnostic modules to be used to analyze the error.

Resource Class General class of the resource that detected the failure (for example, a device class of `disk`).

Resource Type Type of the resource that detected the failure (for example, a device type of `355mb`).

Location Code Path to the device. There may be up to four fields, which refer to drawer, slot, connector, and port, respectively.

VPD Vital product data. The contents of this field, if any, vary. Error log entries for devices typically return information concerning the device manufacturer, serial number, Engineering Change levels, and Read Only Storage levels.

Description Summary of the error.

Probable Cause Listing of some of the possible sources of the error.

User Causes List of possible reasons for errors due to user mistakes. An improperly inserted disk and external devices (such as modems and printers) that are not turned on are examples of user-caused errors.

Recommended Actions	Description of actions for correcting a user-caused error.
Install Causes	List of possible reasons for errors due to incorrect installation or configuration procedures. Examples of this type of error include hardware and software mismatches, incorrect installation of cables or cable connections becoming loose, and improperly configured systems.
Recommended Actions	Description of actions for correcting an installation-caused error.
Failure Causes	List of possible defects in hardware or software. **Note:** A failure causes section in a software error log usually indicates a software defect. Logs that list user or install causes or both, but not failure causes, usually indicate that the problem is not a software defect. If you suspect a software defect, or are unable to correct user or install causes, report the problem to your software service department.
Recommended Actions	Description of actions for correcting the failure. For hardware errors, `PERFORM PROBLEM DETERMINATION PROCEDURES` is one of the recommended actions listed. For hardware errors, this will lead to running the diagnostic programs.
Detailed Data	Failure data that is unique for each error log entry, such as device sense data.

Reporting may be turned off for some errors. To show which errors have reporting turned off, enter:

```
errpt -t -F report=0 | pg
```

If reporting is turned off for any errors, enable reporting of all errors using the **errupdate** command.

Logging may also have been turned off for some errors. To show which errors have logging turned off, enter:

```
errpt -t -F log=0 | pg
```

If logging is turned off for any errors, enable logging for all errors using the **errupdate** command. Logging all errors is useful if it becomes necessary to recreate a system error.

Examples of Detailed Error Reports

The following are sample error report entries that are generated by issuing the **errpt -a** command.

An error-class value of **H** and an error-type value of **PERM** indicate that the system encountered a problem with a piece of hardware (the SCSI adapter device driver) and could not recover from it.

There may be diagnostic data associated with this type of error.

Such information appears at the end of the error's listing.

```
LABEL:      SCSI_ERR1
ID:         0502F666

Date/Time:        Jun 19 22:29:51
Sequence Number:  95
Machine ID:       123456789012
Node ID:          host1
Class:            H
Type:             PERM
Resource Name:    scsi0
Resource Class:   adapter
Resource Type:    hscsi
Location:         00-08
VPD:
     Device Driver Level.........00
     Diagnostic Level............00
```

```
     Displayable Message.........SCSI
     EC Level....................C25928
     FRU Number..................30F8834
     Manufacturer................IBM97F
     Part Number.................59F4566
     Serial Number...............00002849
     ROS Level and ID............24
     Read/Write Register Ptr.....0120

Description
ADAPTER ERROR

Probable Causes
ADAPTER HARDWARE CABLE
CABLE TERMINATOR DEVICE

Failure Causes
ADAPTER
CABLE LOOSE OR DEFECTIVE

          Recommended Actions
          PERFORM PROBLEM DETERMINATION PROCEDURES
          CHECK CABLE AND ITS CONNECTIONS

Detail Data
SENSE DATA
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

Diagnostic Log sequence number:  153
Resource Tested:        scsi0
Resource Description:   SCSI I/O Controller
Location:               00-08
SRN:                    889-191
Description:            Error log analysis indicates hardware failure.
Probable FRUs:
    SCSI Bus        FRU: n/a            00-08
                    Fan Assembly
    SCSI2           FRU: 30F8834        00-08
                    SCSI I/O Controller
```

An error-class value of **H** and an error-type value of **PEND** indicate that a piece of hardware (the Token Ring) may become unavailable soon due to numerous errors detected by the system.

```
LABEL:    TOK_ESERR
ID:       AF1621E8

Date/Time:       Jun 20 11:28:11
Sequence Number: 17262
Machine Id:      123456789012
Node Id:         host1
Class:           H
Type:            PEND
Resource Name:   TokenRing
Resource Class:  tok0
Resource Type:   Adapter
Location:        TokenRing


Description
EXCESSIVE TOKEN-RING ERRORS


Probable Causes
TOKEN-RING FAULT DOMAIN


Failure Causes
TOKEN-RING FAULT DOMAIN


        Recommended Actions
        REVIEW LINK CONFIGURATION DETAIL DATA
        CONTACT TOKEN-RING ADMINISTRATOR RESPONSIBLE FOR THIS LAN
```

```
Detail Data
SENSE DATA
0ACA 0032 A440 0001 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 2080 0000 0000 0010 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 78CC 0000 0000 0005 C88F 0304 F4E0 0000 1000 5A4F 5685
1000 5A4F 5685 3030 3030 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000
```

An error-class value of **S** and an error-type value of **PERM** indicate that the system encountered a problem with software and could not recover from it.

```
LABEL:    DSI_PROC
ID:       20FAED7F

Date/Time:       Jun 28 23:40:14
Sequence Number: 20136
Machine Id:      123456789012
Node Id:         123456789012
Class:           S
Type:            PERM
Resource Name:   SYSVMM

Description
Data Storage Interrupt, Processor

Probable Causes
SOFTWARE PROGRAM

Failure Causes
SOFTWARE PROGRAM

        Recommended Actions
        IF PROBLEM PERSISTS THEN DO THE FOLLOWING
        CONTACT APPROPRIATE SERVICE REPRESENTATIVE

Detail Data
Data Storage Interrupt Status Register
4000 0000
Data Storage Interrupt Address Register
0000 9112
Segment Register, SEGREG
D000 1018
EXVAL
0000 0005
```

An error-class value of **S** and an error-type value of **TEMP** indicate that the system encountered a problem with software. After several attempts, the system was able to recover from the problem.

```
LABEL:          SCSI_ERR6
ID:             52DB7218

Date/Time:       Jun 28 23:21:11
Sequence Number: 20114
Machine Id:      123456789012
Node Id:         host1
Class:           S
Type:            INFO
Resource Name:   scsi0

Description
SOFTWARE PROGRAM ERROR
```

```
Probable Causes
SOFTWARE PROGRAM

Failure Causes
SOFTWARE PROGRAM

        Recommended Actions
        IF PROBLEM PERSISTS THEN DO THE FOLLOWING
        CONTACT APPROPRIATE SERVICE REPRESENTATIVE

Detail Data
SENSE DATA
0000 0000 0000 0000 0000 0011 0000 0008 000E 0900 0000 0000 FFFF
FFFE 4000 1C1F 01A9 09C4 0000 000F 0000 0000 0000 0000 FFFF FFFF
0325 0018 0040 1500 0000 0000 0000 0000 0000 0000 0000 0000 0800
0000 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000
```

An error class value of **O** indicates that an informational message has been logged.

```
LABEL:     OPMSG
ID:        AA8AB241

Date/Time:       Jul 16 03:02:02
Sequence Number: 26042
Machine Id:      123456789012
Node Id:         host1
Class:           O
Type:            INFO
Resource Name:   OPERATOR

Description
OPERATOR NOTIFICATION

User Causes
errlogger COMMAND

        Recommended Actions
        REVIEW DETAILED DATA

Detail Data
MESSAGE FROM errlogger COMMAND
hdisk1 : Error log analysis indicates a hardware failure.
```

Example of a Summary Error Report

The following is an example of a summary error report generated using the **errpt** command. One line of information is returned for each error entry.

```
ERROR_
IDENTIFIER TIMESTAMP  T CL RESOURCE_NAME ERROR_DESCRIPTION
192AC071   0101000070 I 0  errdemon      Error logging turned off
0E017ED1   0405131090 P H  mem2          Memory failure
9DBCFDEE   0101000070 I 0  errdemon      Error logging turned on
038F2580   0405131090 U H  scdisk0       UNDETERMINED ERROR
AA8AB241   0405130990 I O  OPERATOR      OPERATOR NOTIFICATION
```

Generating an Error Report

Use the following procedure to create an error report of software or hardware problems.

1. Determine if error logging is on or off. To do this, determine if the error log contains entries:

```
errpt -a
```

The **errpt** command generates an error report from entries in the system error log.

If the error log does not contain entries, error logging has been turned off. Activate the facility by entering:

```
/usr/lib/errdemon
```

> **Note:** You must have root user access to run this command.

The **errdemon** daemon starts error logging and writes error log entries in the system error log. If the daemon is not running, errors are not logged.

2. Generate an error log report using the **errpt** command. For example, to see all the errors for the `hdisk1` disk drive, enter:

   ```
   errpt -N hdisk1
   ```

3. Generate an error log report using SMIT. For example, use the **smit errpt** command:

   ```
   smit errpt
   ```

   Select **1** to send the error report to standard output or **2** to send the report to the printer.

   Select **yes** to display or print error log entries as they occur; otherwise, select `no`.

   Specify the appropriate device name in the **Select resource names** option (such as `hdisk1`).

   Select **Do**.

Stopping an Error Log

This procedure describes how to stop the error logging facility. Ordinarily, you would not want to turn off the error logging facility. Instead, you should clean the error log of old or unnecessary entries. For instructions about cleaning the error log, refer to “Cleaning an Error Log”.

You should turn off the error logging facility when installing or experimenting with new software or hardware. This way the error logging daemon does not use CPU time to log problems you know you are causing.

> **Note:** You must have root user authority to use the command in this procedure.

Enter the **errstop** command to turn off error logging:

```
errstop
```

The **errstop** command stops the error logging daemon from logging entries.

Cleaning an Error Log

This procedure describes how to strip old or unnecessary entries from your error log. Cleaning is normally done for you as part of the daily **cron** command.

If it is not done automatically, you should probably clean the error log yourself every couple of days after you have examined the contents to make sure there are no significant errors.

You can also clean up specific errors. For example, if you get a new disk and you do not want the old disk’s errors in the log to confuse you, you can clean just the disk errors.

Delete all entries in your error log by doing either of the following:

- Use the **errclear -d** command. For example, to delete all software errors enter:

  ```
  errclear -d S 0
  ```

  The **errclear** command deletes entries from the error log that are older than a certain number of days. The `0` in the previous example means that you want to delete entries for all days.
- Use the **smit errclear** command:

  ```
  smit errclear
  ```

Copying an Error Log to Diskette or Tape

Copy an error log by:

- Use the **ls** and **backup** commands to copy the error log to diskette. Place a formatted diskette into the diskette drive and enter:

  ```
  ls /var/adm/ras/errlog | backup -ivp
  ```
- To copy the error log to tape, place a tape in the drive and enter:

  ```
  ls /var/adm/ras/errlog | backup -ivpf/dev/rmt0
  ```

  OR
- Use the **snap** command to gather system configuration information in a **tar** file and copy it to diskette. Place a formatted diskette into the diskette drive and enter:

  Note: You need root user authority to use the **snap** command.

  ```
  snap -a -o /dev/rfd0
  ```

  The **snap** command in this example uses the **-a** flag to gather all information about your system configuration. The **-o** flag copies the compressed **tar** file to the device you name. `/dev/rfd0` names your disk drive.

  Enter the following command to gather all configuration information in a **tar** file and copy it to tape:

  ```
  snap -a -o /dev/rmt0
  ```

  `/dev/rmt0` names your tape drive.

  See the **snap** command in *AIX 5L Version 5.1 Commands Reference* for more information.

Error Logging and Alerts

If the `Alert` field of an error record template is set to `True`, programs which process alerts use the following fields in the error log to build an alert:

- `Class`
- `Type`
- `Description`
- `Probable Cause`
- `User Cause`
- `Install Cause`
- `Failure Cause`
- `Recommended Action`
- `Detail Data`

These template fields must be set up according to the SNA Generic Alert Architecture described in *SNA Formats*, order number GA27-3136. Alerts that are not set up according to the architecture cannot be processed properly by a receiving program, such as NetView.

Messages added to the error logging message sets must not conflict with the SNA Generic Alert Architecture. When the **errmsg** command is used to add messages, the command selects message numbers that do not conflict with the architecture.

If the `Alert` field of an error record template is set to `False`, you can use any of the messages in the error logging message catalog.

Error Logging Controls

You can control the error logging facility by using the following:

- "Error Logging Commands"
- "Error Logging Subroutines and Kernel Services" on page 99
- "Error Logging Files" on page 99

Error Logging Commands

errclear

Deletes entries from the error log. This command can erase the entire error log. Removes entries with specified error ID numbers, classes, or types.

errdead

Extracts errors contained in the /dev/error buffer captured in the system dump. The system dump will contain error records if the **errdemon** daemon was not active prior to the dump.

errdemon

Reads error records from the **/dev/error** file and writes error log entries to the system error log. The **errdemon** also performs error notification as specified in the error notification objects in the Object Data Manager (ODM). This daemon is started automatically during system initialization.

errinstall

Can be used to add or replace messages in the error message catalog. Provided for use by software installation procedures. The system creates a backup file named *File*.**undo**. The **undo** file allows you to cancel the changes you made by issuing the **errinstall** command.

errlogger

Writes an operator message entry to the error log.

errmsg

Implements error logging in in-house applications. The **errmsg** command lists, adds, or deletes messages stored in the error message catalog. Using this command, text can be added to the `Error Description`, `Probable Cause`, `User Cause`, `Install Cause`, `Failure Cause`, `Recommended Action`, and `Detailed Data` message sets.

errpt

Generates an error report from entries in the system error log. The report can be formatted as a single line of data for each entry, or the report can be a detailed listing of data associated with each entry in the error log. Entries of varying classes and types can be omitted from or included in the report.

errstop

	Stops the **errdemon** daemon, which is initiated during system initialization. Running the **errstop** command also disables some diagnostic and recovery functions of the system.
errupdate	Adds or deletes templates in the Error Record Template Repository. Modifies the Alert, Log, and Report attributes of an error template. Provided for use by software installation procedures.

Error Logging Subroutines and Kernel Services

errlog	Writes an error to the error log device driver.
errsave and **errlast**	Alllows the kernel and kernel extensions to write to the error log.

Error Logging Files

/dev/error	Provides standard device driver interfaces required by the error log component.
/dev/errorctl	Provides nonstandard device driver interfaces for controlling the error logging system.
/usr/include/sys/err_rec.h	Contains structures defined as arguments to the **errsave** kernel service and the **errlog** subroutine.
/var/adm/ras/errlog	Stores instances of errors and failures encountered by the system.
/var/adm/ras/errtmplt	Contains the Error Record Template Repository.

Related Information

The “Error Notification Object Class” in *AIX 5L Version 5.1 General Programming Concepts: Writing and Debugging Programs* allows applications to be notified when particular errors are recorded.

Chapter 5. File Systems and Directories

A file is a one-dimensional array of bytes that can contain ASCII or binary information. In this operating system, files can contain data, shell scripts, and programs. File names are also used to represent abstract objects such as sockets or device drivers.

Internally, files are represented by index nodes (i-nodes). Within this file system, an i-node is a, 128-byte in JFS and 512-byte in JFS2, structure that contains all access, timestamp, ownership, and data location information for each file. Pointers within the i-node structure designate the real disk address of the data blocks associated with the file. An i-node is identified by an offset number (i-number) and has no file name information. The connection of i-numbers and file names is called a link.

File names exist only in directories. Directories are a unique type of file that give hierarchical structure to the file system. Directories contain directory entries. Each directory entry contains a file name and an i-number.

The journaled file system (JFS) and (JFS2) are native to this operating system. The file system links the file and directory data to the structure used by storage and retrieval mechanisms.

JFS and JFS2 are both supported on POWER-based platforms. JFS2 is supported on the Itanium-based platform while JFS is not.

This chapter contains the following sections that further describe the journaled file system programming model:

File Types

A file is a one-dimensional array of bytes with at least one hard link (file name). Files can contain ASCII or binary information. Files contain data, shell scripts, or programs. File names are also used to represent abstract objects such as sockets, pipes, and device drivers.

The kernel does not distinguish record boundaries in regular files. Programs can establish their own boundary markers if desired. For example, many programs use line-feed characters to mark the end of lines. "Working with Files" on page 102 contains a list of the subroutines used to control files.

Files are represented in the journaled file system (JFS and JFS2) by disk index nodes (i-node). Information about the file (such as ownership, access modes, access time, data addresses, and

modification time) is stored in the i-node. For more information about the internal structure of files, see “Working with JFS i-nodes” on page 106 or “Working with JFS2 i-nodes” on page 108.

The journaled file system supports the following file types:

File Types Supported By Journaled File System		
Type of File	**Macro Name Used in mode.h**	**Description**
Regular	**S_ISREG**	A sequence of bytes with one or more names. Regular files can contain ASCII or binary data. These files can be randomly accessed (read from or written to) from any byte in the file.
Directory	**S_ISDIR**	Contains directory entries (file name and i-number pairs). Directory formats are determined by the file system. Processes read directories as they do ordinary files, but the kernel reserves the right to write to a directory. Special sets of subroutines control directory entries.
Block Special	**S_ISBLK**	Associates a structured device driver with a file name.
Character Special	**S_ISCHR**	Associates an unstructured device driver with a file name.
Pipes	**S_ISFIFO**	Designates an interprocess communication channel (IPC). The **mkfifo** subroutine creates named pipes. The **pipe** subroutine creates unnamed pipes.
Symbolic Links	**S_ISLNK**	A file that contains either an absolute or relative path name to another file name.
Sockets	**S_ISSOCK**	An IPC mechanism that allows applications to exchange data. The **socket** subroutine creates sockets, and the **bind** subroutine allows sockets to be named.

The maximum size of a regular file in a JFS file system enabled for large files (available beginning in AIX 4.2) is slightly less than 64 gigabytes (68589453312). All nonregular files in a file system enabled for large files and all files in other JFS file system types have a maximum file size of 2 gigabytes minus 1 (2147483647). The maximun size of a file in JFS2 is limited by the size of the file system itself.

The maximum length of a file name is 255 characters, and the maximum length of a path name is 1023 bytes. For more information, see “JFS File Space Allocation” on page 109.

Working with Files

The operating system offers many subroutines that manipulate files. Brief descriptions of the most common file-control subroutines are provided in two categories:

Creating Files

creat	Creates a new, empty, regular file
open	Creates a new, empty file if the **O_CREAT** flag is set
mkfifo	Creates a named pipe
mkdir	Creates a directory
mknod	Creates a file that defines a device
socket	Creates a socket
pipe	Creates an IPC
link	Creates an additional name (directory entry) for an existing file

Manipulating Files (Programming)

open	Returns a file descriptor used by other subroutines to reference the opened file. The **open** operation takes a regular file name and a permission mode that indicates whether the file is to be read from, written to, or both.
read	Removes data from an open file if the appropriate permissions (**O_RDONLY** or **O_RDWR**) were set by the **open** subroutine.
write	Puts data into an open file if the appropriate permissions (**O_WRONLY** or **O_RDWR**) were set by the **open** subroutine.
lseek or **llseek**	Move the I/O pointer position in an open file.
close	Closes open file descriptors (including sockets).
rmdir	Removes directories from the file system.
chown	Changes ownership of a file.
chmod	Changes the access modes of a file.
stat	Reports the status of a file including the owner and access modes.
access	Determines the accessibility of a file.
rename	Changes the name of a file.
truncate	Changes the length of a file.
ioctl	Controls functions associated with open file descriptors, including special files, sockets, and generic device support like the termio general terminal interface.
fclear	Creates space in file.
fsync	Writes changes in a file to permanent storage.
fcntl, **dup**, or **dup2**	Control open file descriptors.
lockf or **flock**	Control open file descriptors.

For more information on types and characteristics of file systems, see ″File Systems Overview″ in *AIX 5L Version 5.1 System Management Guide: Operating System and Devices*.

JFS Directories

Directories provide a hierarchical structure to the file system and link file and subdirectory names to i-nodes. There is no limit on the depth of nested directories. Disk space is allocated for directories in 4096-byte blocks, but the operating system allocates directory space in 512-byte records.

Processes can read directories as regular files. However, the kernel reserves the right to write directories. For this reason, directories are created and maintained by a set of subroutines unique to them.

JFS Directory Structures

Directories contain a sequence of directory entries. Each directory entry contains three fixed-length fields (the index number associated with the file's i-node, the length of the file name, and the number of bytes for the entry) and one variable length field for the file name. The file name field is null-terminated and padded to 4 bytes. File names can be up to 255 bytes long.

Directory entries are variable-length to allow file names the greatest flexibility. However, all directory space is allocated at all times.

No directory entry is allowed to span 512-byte sections of a directory. When a directory requires more than 512 bytes, another 512-byte record is appended to the original record. If all of the 512-byte records in the allocated data block are filled, an additional data block (4096 bytes) is allotted.

When a file is removed, the space the file occupied in the directory structure is added to the preceding directory entry. The information about the directory remains until a new entry fits into the space vacated.

Every well-formed directory contains the entries . (dot) and .. (dot, dot). The . (dot) directory entry points to the i-node for the directory itself. The .. (dot, dot) directory entry points to the i-node for the parent directory. The **mkfs** program initializes a file system so that the . (dot) and .. (dot, dot) entries in the new root directory point to the root i-node of the file system.

Access modes for directories have the following meanings:

read	Allows a process to read directory entries
write	Allows a process to create new directory entries or remove old ones by using the **creat**, **mknod**, **link**, and **unlink** subroutines
execute	Allows a process to use the directory as a current working directory or to search below the directory in the file tree

Working with Directories (Programming)

The **mkdir** and **rmdir** subroutines create and remove directories, respectively.

The **opendir**, **readdir**, **telldir**, **seekdir**, **rewinddir** and **closedir** subroutines manipulate directories. The **opendir** subroutine returns a structure pointer that is used by the **readdir** subroutine to obtain the next directory entry, by **rewinddir** to reset the read position to the beginning, and by **closedir** to close the directory. The **seekdir** subroutine returns to a position previously obtained with the **telldir** subroutine. In earlier versions, programs treated directories as regular files and used the **open**, **read**, **lseek**, and **close** subroutines to access them. This is no longer recommended.

Changing Current Directory of a Process

When the system is booted, the first process uses the root directory of the root file system as its current directory. New processes created with the **fork** subroutine inherit the current directory used by the parent process. The **chdir** subroutine changes the current directory of a process.

The **chdir** subroutine parses the path name to ensure that the target file is a directory and that the process owner has permissions to the directory. After the **chdir** subroutine is run, the process uses the new current directory to search all path names that do not begin with a / (slash).

Changing the Root Directory of a Process

Processes can change their understanding of the root directory through the **chroot** subroutine. Child processes of the calling process consider the directory indicated by the **chroot** subroutine as the logical root directory of the file system.

Processes use the global file system root directory for all path names starting with a / (slash).All path name searches beginning with a / (slash) begin at this new root directory.

Subroutines That Control Directories

Due to the unique nature of directory files, directories are controlled by a special set of subroutines. The following subroutines are designed to control directories:

chdir	Changes the current working directory
chroot	Changes the effective root directory
opendir, **readdir**, **telldir**, **seekdir**, **rewinddir**, or **closedir**	Perform various actions on directories
getcwd or **getwd**	Gets path to current directory
mkdir	Creates a directory
rename	Renames a directory
rmdir	Removes a directory

JFS2 Directories

Directories provide a hierarchical structure to the file system and link file and subdirectory names to i-nodes. There is no limit on the depth of nested directories. Disk space is allocated for directories in blocks.

Processes can read directories as regular files. However, the kernel reserves the right to write directories. For this reason, directories are created and maintained by a set of subroutines unique to them.

JFS2 Directory Structures

A directory contains entries which indicate the objects contained in the directory. A directory entry has a fixed length. It contains the i-node number, the name up to 22 bytes long, a name length field, and a field to continue the entry if the name won't fit completely.

The directory entries are stored in a B+ tree sorted by name. The self (.) and parent (..) information will be contained in the i-node instead of a directory entry.

Access modes for directories have the following meanings:

read	Allows a process to read directory entries
write	Allows a process to create new directory entries or remove old ones by using the **creat**, **mknod**, **link**, and **unlink** subroutines
execute	Allows a process to use the directory as a current working directory or to search below the directory in the file tree

Working with Directories (Programming)

The **mkdir** and **rmdir** subroutines create and remove directories, respectively.

The **opendir**, **readdir**, **telldir**, **seekdir**, **rewinddir** and **closedir** subroutines manipulate directories. The **opendir** subroutine returns a structure pointer that is used by the **readdir** subroutine to obtain the next directory entry, by **rewinddir** to reset the read position to the beginning, and by **closedir** to close the directory. The **seekdir** subroutine returns to a position previously obtained with the **telldir** subroutine. In earlier versions, programs treated directories as regular files and used the **open**, **read**, **lseek**, and **close** subroutines to access them. This is no longer recommended.

Changing Current Directory of a Process

When the system is booted, the first process uses the root directory of the root file system as its current directory. New processes created with the **fork** subroutine inherit the current directory used by the parent process. The **chdir** subroutine changes the current directory of a process.

The **chdir** subroutine parses the path name to ensure that the target file is a directory and that the process owner has permissions to the directory. After the **chdir** subroutine is run, the process uses the new current directory to search all path names that do not begin with a / (slash).

Changing the Root Directory of a Process

Processes can change their understanding of the root directory through the **chroot** subroutine. Child processes of the calling process consider the directory indicated by the **chroot** subroutine as the logical root directory of the file system.

Processes use the global file system root directory for all path names starting with a / (slash).All path name searches beginning with a / (slash) begin at this new root directory.

Subroutines That Control Directories

Due to the unique nature of directory files, directories are controlled by a special set of subroutines. The following subroutines are designed to control directories:

chdir	Changes the current working directory
chroot	Changes the effective root directory
opendir, **readdir**, **telldir**, **seekdir**, **rewinddir**, or **closedir**	Perform various actions on directories
getcwd or **getwd**	Gets path to current directory
mkdir	Creates a directory
rename	Renames a directory
rmdir	Removes a directory

Working with JFS i-nodes

Files in the journaled file system (JFS) are represented internally as index nodes (i-nodes). Journaled file system i-nodes exist in a static form on disk and contain access information for the file as well as pointers to the real disk addresses of the file's data blocks. The number of disk i-nodes available to a file system is dependent on the size of the file system, the allocation group size (8 MB by default), and the number of bytes per i-node ratio (4096 by default). These parameters are given to the **mkfs** command at file system creation. When enough files have been created to use all the available i-nodes, no more files can be created, even if the file system has free space. The number of available i-nodes can be determined by using the **df -v** command. Disk i-nodes are defined in the **/usr/include/jfs/ino.h** file.

When a file is opened, an in-core i-node is created by the operating system. The in-core i-node contains a copy of all the fields defined in the disk i-node, plus additional fields for tracking the in-core i-node. In-core i-nodes are defined in the **/usr/include/jfs/inode.h** file.

Disk i-node Structure for JFS

Each disk i-node in the journaled file system (JFS) is a 128-byte structure.

The offset of a particular i-node within the i-node list of the file system produces the unique number (i-number) by which the operating system identifies the i-node. A bit map, known as the i-node map, tracks the availability of free disk i-nodes for the file system.

Disk i-nodes include the following information:

Field	Contents
`i_mode`	Type of file and access permission mode bits
`i_size`	Size of file in bytes
`i_uid`	Access permissions for the user ID

Field	Contents
i_gid	Access permissions for the group ID
i_nblocks	Number of blocks allocated to the file
i_mtime	Last time file was modified
i_atime	Last time file was accessed
i_ctime	Last time i-node was modified
i_nlink	Number of hard links to the file
i_rdaddr[8]	Real disk addresses of the data
i_rindirect	Real disk address of the indirect block, if any

It is impossible to change the data of a file without changing the i-node, but it is possible to change the i-node without changing the contents of the file. For example, when permission is changed, the information within the i-node (i_ctime) is modified, but the data in the file remains the same.

The i_rdaddr field within the disk i-node contains 8 disk addresses. These addresses point to the first 8 data blocks assigned to the file. The i_rindirect field address points to an indirect block. Indirect blocks are either single indirect or double indirect. Thus, there are three possible geometries of block allocation for a file: direct, indirect, or double indirect. Use of the indirect block and other file space allocation geometries are discussed in the article “JFS File Space Allocation” on page 109.

Disk i-nodes do not contain file or path name information. Directory entries are used to link file names to i-nodes. Any i-node can be linked to many file names by creating additional directory entries with the **link** or **symlink** subroutine. To discover the i-node number assigned to a file, use the **ls -i** command.

The i-nodes that represent files that define devices contain slightly different information from i-nodes for regular files. Files associated with devices are called special files. There are no data block addresses in special device files, but the major and minor device numbers are included in the i_rdev field.

In normal situations, a disk i-node is released when the link count (i_nlink) to the i-node equals 0. Links represent the file names associated with the i-node. When the link count to the disk i-node is 0, all the data blocks associated with the i-node are released to the bit map of free data blocks for the file system. The i-node is then placed on the free i-node map.

In-core i-node Structure

When a file is opened, the information in the disk i-node is copied into an in-core i-node for easier access. The in-core i-node structure contains additional fields which manage access to the disk i-node’s valuable data. The fields of the in-core i-node are defined in the **inode.h** file. Some of the additional information tracked by the in-core i-node is:

- Status of the in-core i-node, including flags that indicate:
  - An i-node lock
  - A process waiting for the i-node to unlock
  - Changes to the file’s i-node information
  - Changes to the file’s data
- Logical device number of the file system that contains the file
- i-number used to identify the i-node
- Reference count. When the reference count field equals 0, the in-core i-node is released.

When an in-core i-node is released (for instance with the **close** subroutine), the in-core i-node reference count is reduced by 1. If this reduction results in the reference count to the in-core i-node becoming 0, the i-node is released from the in-core i-node table, and the contents of the in-core i-node are written to the disk copy of the i-node (if the two versions differ).

Working with JFS2 i-nodes

Files in the enhanced journaled file system (JFS2) are represented internally as index nodes (i-nodes). JFS2 i-nodes exist in a static form on the disk and they contain access information for the files as well as pointers to the real disk addresses of the file's data blocks. The i-nodes are allocated dynamically by JFS2.

When a file is opened, an in-core i-node is created by the operating system. The in-core i-node contains a copy of all the fields defined in the disk i-node, plus additional fields for tracking the in-core i-node. In-core i-nodes are defined in the /usr/include/j2/j2_inode.h file.

Disk i-node Structure for JFS2

Each disk i-node in JFS2 is a 512 byte structure. The index of a particular i-node allocation map of the file system produces the unique number (i-number) by which the operating system identifies the i-node. The i-node allocation map tracks the location of the i-nodes on the disk as well as their availability.

Disk i-nodes include the following information:

Field	Contents
di_mode	Type of file and access permission mode bits
di_size	Size of file in bytes
di_uid	Access permissions for the user ID
di_gid	Access permissions for the group ID
di_nblocks	Number of blocks allocated to the file
di_mtime	Last time file was modified
di_atime	Last time file was accessed
di_ctime	Last time i-node was modified
di_nlink	Number of hard links to the file
di_btroot	Root of B+ tree describing the disk addresses of the data

It is impossible to change the data of a file without changing the i-node, but it is possible to change the i-node without changing the contents of the file. For example, when permission is changed, the information within the i-node (di_mode) is modified, but the data in the file remains the same.

The di_btroot describes the root of the B+ tree. It describes the data for the i-node. di_btroot has a field indicating how many of its entries in the i-node are being used and another field describing whether they are leaf nodes or internal nodes for the B+ tree. File space allocation geometries are discussed in the article "JFS2 File Space Allocation" on page 113.

Disk i-nodes do not contain file or path name information. Directory entries are used to link file names to i-nodes. Any i-node can be linked to many file names by creating additional directory entries with the **link** or **symlink** subroutine. To discover the i-node number assigned to a file, use the **ls -i** command.

The i-nodes that represent files that define devices contain slightly different information from i-nodes for regular files. Files associated with devices are called special files. There are no data block addresses in special device files, but the major and minor device numbers are included in the di_rdev field.

In normal situations, a disk i-node is released when the link count (di_nlink) to the i-node equals 0. Links represent the file names associated with the i-node. When the link count to the disk i-node is 0, all the data blocks associated with the i-node are released to the bit map of free data blocks for the file system. The i-node is then placed on the free i-node map.

In-core i-node Structure

When a file is opened, the information in the disk i-node is copied into an in-core i-node for easier access. The in-core i-node structure contains additional fields which manage access to the disk i-node's valuable data. The fields of the in-core i-node are defined in the **j2_inode.h** file. Some of the additional information tracked by the in-core i-node is:

- Status of the in-core i-node, including flags that indicate:
  - An i-node lock
  - A process waiting for the i-node to unlock
  - Changes to the file's i-node information
  - Changes to the file's data
- Logical device number of the file system that contains the file
- i-number used to identify the i-node
- Reference count. When the reference count field equals 0, the in-core i-node is released.

When an in-core i-node is released (for instance with the **close** subroutine), the in-core i-node reference count is reduced by 1. If this reduction results in the reference count to the in-core i-node becoming 0, the i-node is released from the in-core i-node table, and the contents of the in-core i-node are written to the disk copy of the i-node (if the two versions differ).

JFS File Space Allocation

File space allocation is the method by which data is apportioned physical storage space in the operating system. The kernel allocates disk space to a file or directory in the form of logical blocks. A *Logical block* refers to the division of a file or directory's contents into 4096 bytes units. Logical blocks are not tangible entities; however, the data in a logical block consumes physical storage space on the disk. Each file or directory consists of 0 or more logical blocks. Fragments, instead of logical blocks, are the basic units for allocated disk space in the journaled file system (JFS).

Full and Partial Logical Blocks

A file or directory may contain full or partial logical blocks. A full logical block contains 4096 bytes of data. Partial logical blocks occur when the last logical block of a file or directory contains less than 4096 bytes of data.

For example, a file of 8192 bytes is two logical blocks. The first 4096 bytes reside in the first logical block and the following 4096 bytes reside in the second logical block. Likewise, a file of 4608 bytes consists of two logical blocks. However, the last logical block is a partial logical block containing the last 512 bytes of the file's data. Only the last logical block of a file can be a partial logical block.

Allocation in Fragmented File Systems

The default fragment size is 4096 bytes. You can specify smaller fragment sizes with the mkfs command during a file system's creation. Allowable fragment sizes are: 512, 1024, 2048, and 4096 bytes. You can use only one fragment size in a file system. See "JFS File System Layout" on page 137 for more information on the file system structure.

To maintain efficiency in file system operations, the JFS allocates 4096 bytes of fragment space to files and directories that are 32KB or larger. A fragment that covers 4096 bytes of disk space is allocated to a full logical block. When data is added to a file or directory, the kernel allocates disk fragments to store the logical blocks. Thus, if the file system's fragment size is 512 bytes, a full logical block is the allocation of 8 fragments.

The kernel allocates disk space so that only the last bytes of data receive a partial block allocation. As the partial block grows beyond the limits of its current allocation, additional fragments are allocated. If the partial block increases to 4096 bytes, the data stored in its fragments are reallocated into 4096 file system block allocations. A partial logical block that contains less than 4096 bytes of data is allocated the number of fragments that best matches its storage requirements.

Block reallocation also occurs if data is added to logical blocks that represent file holes. A *file hole* is an *"empty"* logical block located prior to the last logical block that stores data. (File holes do not occur within directories.) These empty logical blocks are not allocated fragments. However, as data is added to file holes, allocation occurs. Each logical block that was not previously allocated disk space is allocated 4096 byte of fragment space.

Additional block allocation is not required if existing data in the middle of a file or directory is overwritten. The logical block containing the existing data has already been allocated fragments.

JFS tries to maintain contiguous allocation of a file or directory's logical blocks on the disk. Maintaining contiguous allocation lessens seek time because the data for a file or directory can be accessed sequentially and found on the same area of the disk. However, disk fragments for one logical block are not always contiguous to the disk fragments for another logical block. The disk space required for contiguous allocation may not be available if it has already been written to by another file or directory. An allocation for a single logical block, however, always contains contiguous fragments.

The file system uses a bitmap called the *block allocation map* to record the status of every block in the file system. When the file system needs to allocate a new fragment, it refers to the fragment allocation map to identify which fragments are available. A fragment can only be allocated to a single file or directory at a time.

Allocation in Compressed File Systems

In a file system that supports data compression, directories are allocated disk space. Data compression also applies to regular files and symbolic links whose size is larger than that of their i-nodes.

The allocation of disk space for compressed file systems is the same as that of fragments in fragmented file systems. A logical block is allocated 4096 bytes when it is modified. This allocation guarantees that there will be a place to store the logical block if the data does not compress. The system requires that a write or store operation report an out-of-disk-space condition into a memory-mapped file at a logical block's initial modification. After modification is complete, the logical block is compressed before it is written to a disk. The compressed logical block is then allocated only the number of fragments required for its storage.

In a fragmented file system, only the last logical block of a file (not larger than 32KB) can be allocated less than 4096 bytes. The logical block becomes a partial logical block. In a compressed file system, every logical block can be allocated less than a full block.

A logical block is no longer considered modified after it is written to a disk. Each time a logical block is modified, a full disk block is allocated again, according to the system requirements. Reallocation of the initial full block occurs when the logical block of compressed data is successfully written to a disk.

Allocation in File Systems Enabled for Large Files

Beginning in AIX 4.2, in a file system enabled for large files, the JFS allocates two sizes of fragments for regular files. A *"large"* fragment (32 X 4096) is allocated for logical blocks after the 4 MB boundary, and a 4096 bytes fragment is allocated for logical blocks before the 4 MB boundary. All nonregular files allocate 4096 bytes fragments. This geometry allows a maximum file size of slightly less than 64 gigabytes (68589453312).

A *"large"* fragment is made up of 32 contiguous 4096 bytes fragments. Because of this requirement, it is recommended that file systems enabled for large files have predominantly large files in them. Storing many small files (files less than 4 MB) can cause free-space fragmentation problems. This can cause large allocations to fail with ENOSPC because the file system does not contain 32 contiguous disk addresses.

Disk Address Format

JFS fragment support requires fragment-level addressability. As a result, disk addresses have a special format for mapping where the fragments of a logical block reside on the disk. Fragmented and compressed file systems use the same method for representing disk addresses. Disk addresses are contained in the `i_rdaddr` field of the i-nodes or in the indirect blocks. All fragments referenced in a single address must be contiguous on the disk.

The disk address format consists of two fields, the `nfrags` and `addr` fields. These fields describe the area of disk covered by the address.

`addr`	Indicates which fragment on the disk is the starting fragment
`nfrags`	Indicates the total number of contiguous fragments not used by the address

For example, if the fragment size for the file system is 512 bytes and the logical block is divided into eight fragments, the `nfrags` value is 3, indicating that five fragments are included in the address.

The following examples illustrate possible values for the `addr` and `nfrags` fields for different disk addresses. These values assume a fragment size of 512 bytes, indicating that the logical block is divided into eight fragments.

Address for a single fragment:

```
addr:   143
nfrags: 7
```

This address indicates that the starting location of the data is fragment 143 on the disk. The `nfrags` value indicates that the total number of fragments included in the address is one. The `nfrags` value changes in a file system that has a fragment size other than 512 bytes. To correctly read the `nfrags` value, the system, or any user examining the address, must know the fragment size of the file system.

Address for five fragments:

```
addr:   1117
nfrags: 3
```

In this case, the address starts at fragment number 1117 on the disk and continues for five fragments (including the starting fragment). There are three fragments remaining, as illustrated by the `nfrags` value.

The disk addresses are 32 bits in size. The bits are numbered from 0 to 31. The 0 bit is always reserved. Bits 1 through 3 contain the `nfrags` field. Bits 4 through 31 contain the `addr` field.

Indirect Blocks

The JFS uses the indirect blocks to address the disk space allocated to larger files. Indirect blocks allow the greatest flexibility for file sizes and the fastest retrieval time. The indirect block is assigned using the `i_rindirect` field of the disk i-node. This field allows for three geometries or methods for addressing the disk space:

- Direct
- Single indirect

- Double indirect

Each of these methods uses the same disk address format as compressed and fragmented file systems. Because files larger than 32KB are allocated fragments of 4096 bytes, the `nfrags` field for addresses using the single indirect or double indirect method has a value of 0.

Direct Method
When the direct method of disk addressing is used, each of the eight addresses listed in the `i_rdaddr` field of the disk i-node points directly to a single allocation of disk fragments. The maximum size of a file using direct geometry is 32,768 bytes (32KB), or 8 x 4096 bytes. When the file requires more than 32KB, an indirect block is used to address the file's disk space.

Single Indirect Method
The `i_rindirect` field contains an address that points to either a single indirect block or a double indirect block. When the single indirect disk addressing method is used, the `i_rindirect` field contains the address of an indirect block containing 1024 addresses. These addresses point to the disk fragments for each allocation. Using the single indirect block geometry, the file can be up to 4,194,304 bytes (4MB), or 1024 x 4096 bytes.

Double Indirect Method
The double indirect addressing method uses the `i_rindirect` field to point to a double indirect block. The double indirect block contains 512 addresses that point to indirect blocks, which contain pointers to the fragment allocations. The largest file size that can be used with the double indirect geometry in a file system not enabled for large files is 2,147,483,648 bytes (2GB), or 512(1024 x 4096) bytes.

> **Note:** The maximum file size ("Writing Programs That Access Large Files" on page 115) that the **read** and **write** system calls would allow is 2GB minus 1 (2^{31-1}). When memory map interface is used, 2GB can be addresed.

Beginning in AIX 4.2, file systems enabled for large files allow a maximum file size of slightly less than 64 gigabytes (68589453312). The first single indirect block contains 4096 byte fragments, and all subsequent single indirect blocks contain (32 X 4096) byte fragments. The following produces the maximum file size for file systems enabling large files:

```
(1 * (1024 * 4096)) + (511 * (1024 * 131072))
```

The fragment allocation assigned to a directory is divided into records of 512 bytes each and grows in accordance with the allocation of these records.

Quotas

Disk quotas restrict the amount of file system space any single user or group can monopolize.

quotactl Subroutine that sets limits on both the number of files and the number of disk blocks allocated to each user or group on a file system. Quotas enforce two kinds of limits:

- **hard** Maximum limit allowed. When a process hits its hard limit, requests for more space fail.
- **soft** Practical limit. If a process hits the soft limit, a warning is printed to the user's terminal. The warning is often displayed at login. If the user fails to correct the problem after several login sessions, the soft limit can become a hard limit.

System warnings are designed to encourage users to heed the soft limit. However, the quota system allows processes access to the higher hard limit when more resources are temporarily required.

JFS2 File Space Allocation

File space allocation is the method by which data is apportioned physical storage space in the operating system. The kernel allocates disk space to a file or directory in the form of logical blocks. A *Logical block* refers to the division of a file or directory contents into 512, 1024, 2048, or 4096 byte units. When a JFS2 file system is created the logical block size is specified to be one of 512, 1024, 2048, or 4096 bytes. Logical blocks are not tangible entities; however, the data in a logical block consumes physical storage space on the disk. Each file or directory consists of 0 or more logical blocks.

Full and Partial Logical Blocks

A file or directory may contain full or partial logical blocks. A full logical block contains 512, 1024, 2048, or 4096 bytes of data, depending on the file system block size specified when the JFS2 file system was created. Partial logical blocks occur when the last logical block of a file or directory contains less than file system block size of data.

For example, a JFS2 file system with a logical block size of 4096 with a file of 8192 bytes is two logical blocks. The first 4096 bytes reside in the first logical block and the following 4096 bytes reside in the second logical block. Likewise, a file of 4608 bytes consists of two logical blocks. However, the last logical block is a partial logical block containing the last 512 bytes of the file's data. Only the last logical block of a file can be a partial logical block.

JFS2 File Space Allocation

The default block size is 4096 bytes. You can specify smaller block sizes with the **mkfs** command during a file system's creation. Allowable fragment sizes are: 512, 1024, 2048, and 4096 bytes. You can use only one blocks size in a file system. See "JFS File System Layout" on page 137 for more information on the file system structure.

The kernel allocates disk space so that only the last file system block of data receive a partial block allocation. As the partial block grows beyond the limits of its current allocation, additional blocks are allocated.

Block reallocation also occurs if data is added to logical blocks that represent file holes. A *file hole* is an *"empty"* logical block located prior to the last logical block that stores data. (File holes do not occur within directories.) These empty logical blocks are not allocated blocks. However, as data is added to file holes, allocation occurs. Each logical block that was not previously allocated disk space is allocated a file system block of space.

Additional block allocation is not required if existing data in the middle of a file or directory is overwritten. The logical block containing the existing data has already been allocated file system blocks.

JFS tries to maintain contiguous allocation of a file or directory's logical blocks on the disk. Maintaining contiguous allocation lessens seek time because the data for a file or directory can be accessed sequentially and found on the same area of the disk. The disk space required for contiguous allocation may not be available if it has already been written to by another file or directory.

The file system uses a bitmap called the *block allocation map* to record the status of every block in the file system. When the file system needs to allocate a new block, it refers to the block allocation map to identify which blocks are available. A block can only be allocated to a single file or directory at a time.

Extents

An extent is a sequence of contiguous file system blocks allocated to a JFS2 object as a unit. Large extents may span multiple allocation groups.

Every JFS2 object is represented by an i-node. I-nodes contain the expected object-specific information such as time stamps, file type (regular verses directory etcetera.) They also contain a B+ tree to record the allocation of extents.

A file is allocated in sequences of extents. An extent is a contiguous variable-length sequence of file system blocks allocated as a unit. An extent may span multiple allocation groups. These extents are indexed in a B+ tree.

There are two values needed to define an extent, the length and the address. The length is measured in units of the file system block size. 24-bit value represents the length of an extent, so an extent can range in size from 1 to 2^{24} -1 file system blocks. Therefore the size of the maximum extent depends on the file system block size. The address is the address of the first block of the extent. The address is also in units of file system blocks, it is the block offset from the beginning of the file system.

An extent based file system combined with user-specified file system block size allows JFS2 to not have seperate support for internal fragmentation. The user can configure the file system with a small file system block size (such as 512 bytes) to minimize internal fragmentation for file systems with large numbers of small size files.

In general, the allocation policy for JFS2 tries to maximize contiguous allocation by allowing a minimum number of extents, with each extent as large and contiguous as possible. This allows for larger I/O transfer resulting in improved performance. However in some cases this is not always possible.

B+ Trees

This section describes the B+ tree data structure used for file layout. The discussion shows how generic B+ tree concepts have been adapted specifically for JFS2; it is not a tutorial on B+ tree data structure.

B+ trees were selected to help with performance of reading and writing extents, the most common operations JFS2 will have to do.

An extent allocation descriptor (**xad_t** structure) describes the extent and adds two more fields that are needed for representing files: an offset, describing the logical the logical byte address the extent represents, and a flags field. **The xad_t** structure is defined in /usr/include/j2/j2_xtree.h.

An **xad** structure describes two abstract ranges:

- The physical range of disk blocks. This starts at file system block number addressXAD(xadp) address and extends for lengthXAD(xadp) file system blocks.
- The logical range of bytes within a file. This starts at byte number offsetXAD(xadp)*(file system block size) and extends for lengthXAD(xadp)*(file system block size.)

The physical range and the logical range are both the same number of bytes long. Note that offset is stored in units of file system block size (example, a value of 3) in offset means 3 file system blocks, not three bytes. Extents within a file are always aligned on file system block size boundaries.

There will be one generic B+ tree index structure for all index objects in JFS2 except for directories. The data being indexed will depend on the object. The B+ tree is keyed by the offset of the **xad** of data being described by the tree. The entries are sorted by the offsets of the **xad** structures. An xad structure is an entry in a node of a B+ tree.

The bottom of the second section of a disk inode contains a data dscriptor which tells what is stored in the second half of the inode. The second half of the inode could contain in-line data for the file if it is small enough. If the file data won't fit in the in-line data space for the inode it will be contained in extents and the inode will contain the root node of the B+ tree. The header will indicate how many xad are in use and how many are available. Generally, the inode will contain 8 xad structures for the root of the B+ tree. If

there are 8 or fewer extents for the file, then these 8 xad structures are also a leaf node of the B+ tree. They will describe the extents. Otherwise the 8 xad structures in the inode will point to either the leaves or internal nodes of the B+ tree.

Once all of the available xad structures in the inode are used, the B+ tree must be split. We will allocate 4K of disk space for a leaf node of the B+ tree. A leaf node is logically an array of xad entries with a header. The header points to the first free xad entry in the node, all xad entries following that one are also not allocated. The 8 xad entries are copied from the inode to the leaf node, the header is initialized to point to the 9th entry as the first free entry. Then we will update the root of the B+ tree into the first xad structure of the inode; this xad structure will point to the newly allocated leaf node. The offset for this new xad structure will be the offset of the first entry in the leaf node. The header in the inode will be updated to indicate that now only 1 xad is being used for the B+ tree. The header in the inode also needs to be updated to indicate the inode now contains the pure root of a B+ tree.

As new extents are added to the file, they will continue to be added to the same leaf node in the necessary order. This will continue until the leaf node fills. Once the node fills a new 4K of disk space will be allocated for another leaf node of the B+ tree. The second xad structure from the inode will be set to point to this newly allocated node.

This will continue until all 8 xad structures in the inode are filled, at which time another split of the B+ tree will occur. This split will create internal nodes of the B+ tree which are used purely to route the searches of the tree. This will allocate 4K of disk space for an internal node of the B+ tree. An internal node looks the same as a leaf node. The 8 xad entries are copied from the inode to the internal node, the header is initialized to point to the 9th entry as the first free entry. Then it will update the root of the B+ tree by making the first xad structure of the inode point to the newly allocated internal node. The header in the inode will be updated to indicate that now only 1 xad is being used for the B+ tree.

The file /usr/include/j2/j2_xtree.h describes the header for the root of the B+ tree in **struct xtpage_t**. The file /usr/include/j2/j2_btree.h describes the header for an internal node or a leaf node in **struct btpage_t**.

Writing Programs That Access Large Files

Beginning in AIX 4.2, the operating system allows files that are larger than 2 gigabytes (2GB). This article is intended to assist programmers in understanding the implications of *"large"* files on their applications and to assist them in modifying their applications. A new set of programming interfaces is defined, so that application programs can be modified to be aware of large files.

The file system programming interfaces generally revolve around the off_t data type. In AIX 4.1, the off_t data type was defined as a signed 32-bit integer. As a result, the maximum file size that these interfaces would allow was 2 gigabytes minus 1.

Implications for Existing Programs

The 32-bit application environment that all applications used in prior releases remains unchanged. Existing application programs will execute exactly as they did before. However, existing application programs will not be able to deal with large files.

For example, the `st_size` field in the stat structure, which is used to x turn file sizes, is a signed, 32-bit long. Therefore, that stat structure cannot be used to return file sizes that are larger than LONG_MAX. If an application attempts to **stat** a file that is larger than LONG_MAX, the **stat** subroutine will fail, and errno will be set to EOVERFLOW, indicating that the file size overflows the size field of the structure being used by the program.

This behavior is significant because existing programs that might not appear to have any impacts as a result of large files will experience failures in the presence of large files even though they may not even be interested in the file size.

The errno EOVERFLOW can also be returned by **lseek** and by **fcntl** if the values that need to be returned are larger than the data type or structure that the program is using. For **lseek**, if the resulting offset is larger than LONG_MAX, **lseek** will fail and errno will be set to EOVERFLOW. For **fcntl**, if the caller uses F_GETLK and the blocking lock's starting offset or length is larger than LONG_MAX, the **fcntl** call will fail, and errno will be set to EOVERFLOW.

Open Protection

Many of the existing application programs were written under the assumption that a file size could never be larger than could be represented in a signed, 32-bit long. These programs could have unexpected behavior, including data corruption, if allowed to operate on large files. Beginning in AIX 4.2, the operating system implements an open-protection scheme to protect applications from this class of failure.

When an application that has not been enabled for large-file access attempts to open a file that is larger than LONG_MAX, the **open** subroutine will fail and errno will be set to EOVERFLOW. Application programs that have not been enabled will be unable to access a large file, and the possibility of inadvertent data corruption is avoided. Applications that need to be able to open large files must be ported to the large-file environment described in ″“Porting Applications to the Large File Environment”″.

In addition to open protection, a number of other subroutines offer protection by providing an execution environment, which is identical to the environment under which these programs were developed. If an application uses the **write** family of subroutines and the **write** request crosses the 2 gigabyte boundary, the **write** subroutines will transfer data only up to 2 gigabytes minus 1. If the application attempts to write at or beyond the 2Gb-1 boundary, the **write** subroutines will fail and set errno to EFBIG. The behavior of **mmap**, **ftruncate**, and **fclear** are similar.

The **read** family of subroutines also participates in the open protection scheme. If an application attempts to read a file across the 2 gigabyte threshold, only the data up to 2 gigabytes minus 1 will be read. Reads at or beyond the 2Gb-1 boundary will fail, and errno will be set to EOVERFLOW.

Open protection is implemented by a flag associated with an open file description. The current state of the flag can be queried with the **fcntl** subroutine using the F_GETFL command. The flag can be modified with the **fcntl** subroutine using the F_SETFL command.

Since open file descriptions are inherited across the **exec** family of subroutines, application programs that pass file descriptors that are enabled for large-file access to other programs should consider whether the receiving program can safely access the large file.

Porting Applications to the Large File Environment

Beginning in AIX 4.2, the operating system provides two different ways for applications to be enabled for large-file access. Application programmers must decide which approach best suits their needs. The first approach is to define _LARGE_FILES, which carefully redefines all of the relevant data types, structures, and subroutine names to their large-file enabled counterparts. The second approach is to recode the application to call the large-file enabled subroutines explicitly.

Defining _LARGE_FILES has the advantage of maximizing application portability to other platforms since the application is still written to the normal POSIX and XPG interfaces. It has the disadvantage of creating some ambiguity in the code since the size of the various data items is not obvious from looking at the code.

Recoding the application has the obvious disadvantages of requiring more effort and reducing application portability. It can be used when the redefinition effect of _LARGE_FILES would have a considerable negative impact on the program or when it is desirable to convert only a very small portion of the program.

It is very important to understand that in either case, the application program MUST be carefully audited to ensure correct behavior in the new environment. Some of the common programming pitfalls are discussed in ″“Common Pitfalls using the Large File Environment” on page 119″.

Using _LARGE_FILES

In the default compilation environment, the off_t data type is defined as a signed, 32-bit long. Beginning in AIX 4.2, if the application defines _LARGE_FILES before the inclusion of any header files, then the large-file programming environment is enabled. and off_t is defined to be a signed, 64-bit long long. In addition, all of the subroutines that deal with file sizes or file offsets are redefined to be their large-file enabled counterparts. Similarly, all of the data structures with embedded file sizes or offsets are redefined.

Assuming that the application is coded without any dependencies on off_t being a 32-bit quantity, the resulting binary should work properly in the new environment. In practice, application programs rarely require a porting effort this small.

The following table shows the redefinitions that occur in the _LARGE_FILES environment beginning in AIX 4.2.

Item	Redefined To Be	Header File
off_t	long long	<sys/types.h>
fpos_t	long long	<sys/types.h>
struct stat	struct stat64	<sys/stat.h>
stat()	stat64()	<sys/stat.h>
fstat()	fstat64()	<sys/stat.h>
lstat()	lstat64()	<sys/stat.h>
mmap()	mmap64()	<sys/mman.h>
lockf()	lockf64()	<sys/lockf.h>
struct flock	struct flock64	<sys/flock.h>
open()	open64()	<fcntl.h>
creat()	creat64()	<fcntl.h>
F_GETLK	F_GETLK64	<fcntl.h>
F_SETLK	F_SETLK64	<fcntl.h>
F_SETLKW	F_SETLKW64	<fcntl.h>
ftw()	ftw64()	<ftw.h>
nftw()	nftw64()	<ftw.h>
fseeko()	fseeko64()	<stdio.h>
ftello()	ftello64()	<stdio.h>
fgetpos()	fgetpos64()	<stdio.h>
fsetpos()	fsetpos64()	<stdio.h>
fopen()	fopen64()	<stdio.h>
freopen()	freopen64()	<stdio.h>
lseek()	lseek64()	<unistd.h>

ftruncate()	ftruncate64()	<unistd.h>
truncate()	truncate64()	<unistd.h>
fclear()	fclear64()	<unistd.h>
struct aiocb	struct aiocb64	<sys/aio.h>
aio_read()	aio_read64()	<sys/aio.h>
aio_write()	aio_write64()	<sys/aio.h>
aio_cancel()	aio_cancel64()	<sys/aio.h>
aio_suspend	aio_suspend64()	<sys/aio.h>
aio_listio()	aio_listio64()	<sys/aio.h>
aio_return()	aio_return64()	<sys/aio.h>
aio_error	aio_error64()	<sys/aio.h>

Using the 64-Bit File System Subroutines

Using the _LARGE_FILES environment may be impractical for some applications due to the far-reaching implications of changing the size of off_t to 64 bits. If the number of changes is small, it may be more practical to convert a relatively small part of the application to be large-file enabled. The 64-bit file system data types, structures, and subroutines are listed below:

```
<sys/types.h>
typedef long long off64_t;
typedef long long fpos64_t;

<fcntl.h>

extern int      open64(const char *, int, ...);
extern int      creat64(const char *, mode_t);

#define F_GETLK64
#define F_SETLK64
#define F_SETLKW64

<ftw.h>
extern int ftw64(const char *, int (*)(const char *,const struct stat64 *, int), int);
extern int nftw64(const char *, int (*)(const char *, const struct stat64 *, int,struct FTW *),int, int);

<stdio.h>

extern int      fgetpos64(FILE *, fpos64_t *);
extern FILE     *fopen64(const char *, const char *);
extern FILE     *freopen64(const char *, const char *, FILE *);
extern int      fseeko64(FILE *, off64_t, int);
extern int      fsetpos64(FILE *, fpos64_t *);
extern off64_t  ftello64(FILE *);

<unistd.h>

extern off64_t  lseek64(int, off64_t, int);
extern int      ftruncate64(int, off64_t);
extern int      truncate64(const char *, off64_t);
extern off64_t  fclear64(int, off64_t);

<sys/flock.h>

struct flock64;

<sys/lockf.h>
```

```
extern int lockf64 (int, int, off64_t);

<sys/mman.h>

extern void     *mmap64(void *, size_t, int, int, int, off64_t);

<sys/stat.h>

struct stat64;

extern int      stat64(const char *, struct stat64 *);
extern int      fstat64(int, struct stat64 *);
extern int      lstat64(const char *, struct stat64 *);

<sys/aio.h>

struct aiocb64
int     aio_read64(int, struct aiocb64 *):
int     aio_write64(int, struct aiocb64 *);
int     aio_listio64(int, struct aiocb64 *[],
        int, struct      sigevent *);
int     aio_cancel64(int, struct aiocb64 *);
int     aio_suspend64(int, struct aiocb64 *[]);
```

Common Pitfalls using the Large File Environment

Porting of application programs to the large-file environment can expose a number of different problems in the application. These problems are frequently the result of poor coding practices, which are harmless in a 32-bit off_t environment, but which can manifest themselves when compiled in a 64-bit off_t environment. The information below illustrates some of the more common problems and solutions.

Note: In the examples below, off_t is assumed to be a 64-bit file offset.

Improper Use of Data Types

The most obvious source of problems with application programs is a failure to use the proper data types. If an application attempts to store file sizes or file offsets in an integer variable, the resulting value will be truncated and lose significance. The proper technique for avoiding this problem is to use the off_t data type to store file sizes and offsets.

Incorrect:

```
int file_size;
struct stat s;

file_size = s.st_size;
```

Better:

```
off_t file_size;
struct stat s;
file_size = s.st_size;
```

Parameter Mismatches

Care must be taken when passing 64-bit integers to functions as arguments or when returning 64-bit integers from functions. Both the caller and the called function must agree on the types of the arguments and the return value in order to get correct results.

Passing a 32-bit integer to a function that expects a 64-bit integer causes the called function to misinterpret the caller's arguments, leading to unexpected behavior. This type of problem is especially severe if the program passes scalar values to a function that expects to receive a 64-bit integer.

Many of the problems can be avoided by careful use of function prototypes as illustrated below. In the code fragments below, **fexample()** is a function that takes a 64-bit file offset as a parameter. In the first example, the compiler generates the normal 32-bit integer function linkage, which would be incorrect since the receiving function expects 64-bit integer linkage. In the second example, the LL specifier is added, forcing the compiler to use the proper linkage. In the last example, the function prototype causes the compiler to promote the scalar value to a 64-bit integer. This is the preferred approach since the source code remains portable between 32- and 64-bit environments.

Incorrect:

```
fexample(0);
```

Better:

```
fexample(0LL);
```

Best:

```
void fexample(off_t);

fexample(0);
```

Arithmetic Overflows

Even when an application uses the correct data types, it is still vulnerable to failures due to arithmetic overflows. This problem usually occurs when the application performs an arithmetic overflow before it is promoted to the 64-bit data type. In the following example, blkno is a 32-bit block number. Multiplying the block number by the block size occurs before the promotion, and overflow will occur if the block number is sufficiently large. This problem is especially destructive because the code is using the proper data types and the code works properly for small values, but fails for large values. The problem can be fixed by typecasting the values before the arithmetic operation.

Incorrect:

```
int blkno;
off_t offset;

offset = blkno * BLKSIZE;
```

Better:

```
int blkno;
off_t offset;
offset = (off_t) blkno * BLKSIZE;
```

This problem can also appear when passing values based on fixed constants to functions that expect 64-bit parameters. In the example below, LONG_MAX+1 results in a negative number, which is sign-extended when it is passed to the function.

Incorrect:

```
void fexample(off_t);

 fexample(LONG_MAX+1);
```

Better:

```
void fexample(off_t);

fexample((off_t)LONG_MAX+1);
```

Fseek/Ftell

The data type used by **fseek** and **ftell** subroutines is long and cannot be redefined to the appropriate 64-bit data type in the _LARGE_FILES environment. Application programs that access large files and that

use **fseek** and **ftell** need to be converted. This can be done in a number of ways. The **fseeko** and **ftello** subroutines are functionally equivalent to **fseek** and **ftell** except that the offset is given as an off_t instead of a long. Make sure to convert all variables that can be used to store offsets to the appropriate type.

Incorrect:

```
long cur_offset, new_offset;

cur_offset = ftell(fp);
fseek(fp, new_offset, SEEK_SET);
```

Better:

```
off_t cur_offset, new_offset;

cur_offset = ftello(fp);
fseeko(fp, new_offset, SEEK_SET);
```

Failure to Include Proper Header Files

In order for application programs to see the function and data type redefinitions, they must include the proper header files. This has the additional benefit of exposing the function prototypes for various subroutines, which enables stronger type-checking in the compiler.

Many application programs that call the **open** and **creat** subroutines do not include **<fcntl.h>**, which contains the defines for the various open modes. These programs typically hard code the open modes. This will cause runtime failures when the program is compiled in the _LARGE_FILES environment because the program does call the proper **open** subroutine, and the resulting file descriptor is not enabled for large-file access. Programs must make sure to include the proper header files, especially in the _LARGE_FILES environment, to get visibility to the redefinitions of the environment.

Incorrect:

```
fd = open("afile",2);
```

Better:

```
#include <fcntl.h>

fd = open("afile",O_RDWR);
```

String Conversions

Converting file sizes and offsets to and from strings can cause problems when porting applications to the large-file environment. The **printf** format string for a 64-bit integer is different than for a 32 bit integer. Programs that do these conversions must be careful to use the proper format specifier. This is especially difficult when the application needs to be portable between 32- and 64-bit environments since there is no portable format specifier between the two environments. One way to deal with this problem is to write offset converters that use the proper format for the size of off_t.

```
off_t
atooff(const char *s)
{
        off_t o;

        if (sizeof(off_t) == 4)
                sscanf(s,"%d",&o);
        else if (sizeof(off_t) == 8)
                sscanf(s,"%lld",&o);
        else
                error();
        return o;
}
```

```
        main(int argc, char **argv)
{
        off_t offset;
        offset = atooff(argv[1]);
        fexample(offset);
}
```

Imbedded File Offsets

Application programs that imbed file offsets or sizes in data structures may be affected by the change to the size of the off_t in the large-file environment. This problem can be especially severe if the data structure is shared between various applications or if the data structure is written into a file. In cases like this, the programmer must decide if it should continue to contain a 32-bit offset or if it should be converted to contain a 64-bit offset. If the application program needs to have a 32-bit file offset even if off_t is 64 bits, the program may use the new data type soff_t, a short off_t. This data type remains 32 bits even in the large-file environment. If the data structure is converted to a 64-bit offset, then all of the programs that deal with that structure must be converted to understand the new data structure format.

File Size Limits

Application programs that are converted to be aware of large files may fail in their attempts to create large files due to the file-size resource limit. The file-size resource limit is a signed, 32-bit value which limits maximum file offset to which a process can write to a regular file. Programs that need to write large files must have their file size limit set to RLIM_INFINITY.

```
struct rlimit r;

r.rlim_cur = r.rlim_max = RLIM_INFINITY;
setrlimit(RLIMIT_FSIZE,&r);
```

This limit may also be set from the Korn shell by issuing the command:

```
ulimit -f unlimited
```

To set this value permanently for a specific user, use the **chuser** command:

Example: `chuser fsize_hard = -1 root`

JFS File Size Limits

The maximum size of a file is ultimately a characteristic of the file system itself, not just the file size limit or the environment. For the JFS, the maximum file size is determined by the parameters used at the time the file system was made. For JFS file systems that are enabled for large files, the maximum file size is slightly less than 64 gigabytes (0xff8400000). For all other JFS file systems, the maximum file size is 2Gb-1 (0x7fffffff). Attempts to write a file past the maximum file size in any file system format will fail, and errno will be set to EFBIG.

JFS2 File Size Limits

For the JSF2. the maximun file size is limited by the file system itself.

Linking for Programmers

A link is a connection between a file name and an i-node (hard link) or between file names (symbolic link). Linking allows access to an i-node (“Working with JFS i-nodes” on page 106) from multiple file names. Directory entries pair file names with i-nodes. File names are easy for users to identify, and i-nodes contain the real disk addresses of the file’s data. A reference count of all links into an i-node is maintained in the `i_nlink` field of the i-node. Subroutines that create and destroy links use file names, not file descriptors (“Using File Descriptors” on page 125). Therefore, it is not necessary to open files when creating a link.

Processes can access and change the contents of the i-node by any of the linked file names. Two kinds of links exist in this operating system, hard links and symbolic links.

Hard Links

link Subroutine that creates hard links.The presence of a hard link guarantees the existence of a file because a hard link increments the link count in the `i_nlink` field of the i-node.

unlink Subroutine that releases links. When all hard links to an i-node are released, the file is no longer accessible.

The user ID that created the original file owns the file and retains access mode authority over the file. Otherwise, all hard links are treated equally by the operating system. Hard links must link file names and i-nodes within the same file system since the i-node number is relative to a single file system.

Hard links always refer to a specific file because the directory entry created by the hard link pairs the new file name to an i-node.

Example: If the `/u/tom/bob` file is linked to the `/u/jack/foo` file, the link count in the `i_nlink` field of the `foo` file is `2`. Both hard links are equal. If `/u/jack/foo` is removed, the file continues to exist by the name `/u/tom/bob` and can be accessed by users with access to the `tom` directory. However, the owner of the file is `jack` even though `/u/jack/foo` was removed. The space occupied by the file is charged to jack's quota account. Change file ownership using the **chown** subroutine.

Symbolic Links

symlink Subroutine that creates symbolic links

Symbolic links are implemented as a file that contains a path name. When a process encounters a symbolic link, the path contained in the symbolic link is prepended to the path the process was searching. If the path name in the symbolic link is an absolute path name, the process searches from the root directory for the named file. If the path name in the symbolic link does not begin with a / (slash), the process interprets the rest of the path relative to the position of the symbolic link. The **unlink** subroutine also removes symbolic links.

Symbolic links can traverse file systems because they are treated as regular files by the operating system rather than as part of the file system structure. The presence of a symbolic link does not guarantee the existence of the target file because a symbolic link has no effect on the `i_nlink` field of the i-node.

readlink Subroutine that reads the contents of a symbolic link. Many subroutines (including the **open** and **stat** subroutines) follow symbolic paths.

lstat Subroutine created to report on the status of the file containing the symbolic link and does not follow the link. See the **symlink** subroutine for a list of subroutines that traverse symbolic links.

Symbolic links are also called soft links because they link to a file by path name. If the target file is renamed or removed, the symbolic link cannot resolve.

Example: The symbolic link to `/u/joe/foo` is a file that contains the literal data `/u/joe/foo`. When the owner of the `foo` file removes this file, subroutine calls made to the symbolic link cannot succeed. If the file owner then creates a new file named `foo` in the same directory, the symbolic link leads to the new file. Therefore, the link is considered soft because it is linked to interchangeable i-nodes.

In the **ls -l** command listing, an `l` in the first position indicates a linked file. In the final column of that listing, the links between files are represented as `Path2 -> Path1` (or `Newname -> Oldname`).

unlink Subroutine that removes a directory entry. The *Path* parameter in the subroutine identifies the file to be disconnected. At the completion of the **unlink** call, the link count of the i-node is reduced by the value of 1.

remove Subroutine that also removes a file name by calling either the **unlink** or **rmdir** subroutine.

Directory Links

mkdir Subroutine that creates directory entries for new directories, which creates hard links to the i-node representing the directory

Symbolic links are recommended for creating additional links to a directory. Symbolic links do not interfere with the . and .. directory entries and will maintain the empty, well-formed directory status. See the Understanding Directory Links ″figure″ for a graphic example of the empty, well-formed directory **/u/joe/foo** and the i_nlink values.

```
/u
```

68			j	o	e	0

```
/u/joe
mkdir ("foo", 0666)
```

68			n	0	0	0
			n	n	0	0
235			f	o	o	0

```
/u/joe/foo
```

235			n	0	0	0
68			n	n	0	0

```
i_nlink Values
```

i = 68
n_link 3

For i = 68, the n_link value is 3 (**/u; /u/joe; /u/joe/foo**).

i = 235
n_link 2

For i = 235, the n_link value is 2 (**/u/joe; /u/joe/foo**).

Understanding Directory Links

rmdir or **remove** Remove links to directories

Using File Descriptors

A file descriptor is an unsigned integer used by a process to identify an open file. Two thousand file descriptors are available to each process. The **open**, **pipe**, **creat**, and **fcntl** subroutines all generate file descriptors. File descriptors are generally unique to each process, but they can be shared by child processes created with a **fork** subroutine or copied by the **fcntl**, **dup**, and **dup2** subroutines.

File descriptors are indexes to the file descriptor table in the **u_block** area maintained by the kernel for each process. The most common ways for processes to obtain file descriptors are through **open** or **creat** operations or through inheritance from a parent process. When a **fork** operation occurs, the descriptor table is copied for the child process, which allows the child process equal access to the files used by the parent process.

System File and File Descriptor Tables

The system file and file descriptor data structures track each process' access to a file and ensure data integrity.

Structure	Activity and Contents

file descriptor table	Translates an index number (file descriptor) in the table to an open file. File descriptor tables are created for each process and are located in the **u_block** area set aside for that process. Each of the entries in a file descriptor table has two fields: the flags area and the file pointer. The structure of the file descriptor table is:
	`struct ufd`
	`{`
	`  struct file *fp;`
	`  int flags;`
	`}u_ufd[OPEN_MAX]`
	
	The close-on-exec (**FD_CLOEXEC** bit) flag can be set in the file descriptor table using the **fcntl** subroutine. The **dup** subroutine copies one file descriptor entry into another position in the same table. The **fork** subroutine creates an identical copy of the entire file descriptor table for a child process.

system open file table	Contains entries for each open file. Two of the most important pieces of information tracked in a file table entry are the current offset referenced by all read or write operations to the file and the open mode (**O_RDONLY**, **O_WRONLY**, or **O_RDWR**) of the file. The open file data structure contains the current I/O offset for the file. The system treats each read/write operation as an implied seek to the current offset. Thus if *x* bytes are read or written, the pointer advances *x* bytes. The **lseek** subroutine can be used to reassign the current offset to a specified location in files that are randomly accessible. Stream-type files (such as pipes and sockets) do not use the offset because the data in the file is not randomly accessible.

Managing File Descriptors

Because files can be shared by many users, it is necessary to allow related processes to share a common offset pointer and have a separate current offset pointer for independent processes that access the same file. The open file table entry maintains a reference count to track the number of file descriptors assigned to the file.

Multiple references to a single file can be caused by:

- A separate process opening the file
- Child processes retaining the file descriptors assigned to the parent process
- The **fcntl** or **dup** subroutine creating copies of the file descriptors

Sharing Open Files

Each open operation creates a system table entry. Individual table entries ensure each process a separate current I/O offsets. Independent offsets protect the integrity of the data.

When a file descriptor is duplicated, two processes then share the same offset and interleaving can occur. Interleaving means that bytes are not read or written sequentially.

Duplicating File Descriptors

There are three ways file descriptors can be duplicated between processes: the **dup** or **dup2** subroutine, the **fork** subroutine, and the **fcntl** (file descriptor control) subroutine.

The dup and dup2 Subroutines:

dup Creates a copy of a file descriptor

The duplicate is created at an empty space in the user file descriptor table that contains the original descriptor. A **dup** process increments the reference count in the file table entry by 1 and returns the index number of the file-descriptor where the copy was placed.

dup2 Scans for the requested descriptor assignment and closes the requested file descriptor if it is open

The **dup2** subroutine allows the process to designate which descriptor entry the copy will occupy, if a specific descriptor-table entry is required.

The fork Subroutine:

fork Creates a child process that inherits the file descriptors assigned to the parent process. The child process then execs a new process. Inherited descriptors that had the **close-on-exec** flag set by the **fcntl** subroutine close.

The fcntl (File Descriptor Control) Subroutine:

fcntl Manipulates file structure and controls open file descriptors.

The **fcntl** subroutine can be used to make the following changes to a descriptor:

- Duplicate a file descriptor (identical to the **dup** subroutine).
- Get or set the close-on-exec flag.
- Set nonblocking mode for the descriptor.
- Append future writes to the end of the file (**O_APPEND**).
- Enable the generation of a signal to the process when it is possible to do I/O.
- Set or get the process ID or the group process ID for **SIGIO** handling.
- Close all file descriptors.

Preset File Descriptor Values

When the shell runs a program, it opens three files with file descriptors 0, 1, and 2. The default assignments for these descriptors are:

0 Represents standard input.
1 Represents standard output.
2 Represents standard error.

These default file descriptors are connected to the terminal, so that if a program reads file descriptor 0 and writes file descriptors 1 and 2, the program collects input from the terminal and sends output to the terminal. As the program uses other files, file descriptors are assigned in ascending order.

If I/O is redirected using the < (less than) or > (greater than) symbols, the shell's default file descriptor assignments are changed. For instance:

```
prog < FileX > FileY
```

changes the default assignments for file descriptors 0 and 1 from the terminal to the appropriate files. In this example, file descriptor 0 now refers to `FileX` and file descriptor 1 refers to `FileY`. File descriptor 2 has not been changed. The program does not need to know where its input comes from nor where it is sent, as long as file descriptor 0 represents the input file and 1 and 2 represent output files.

The following sample program illustrates the redirection of standard output:

```
#include <fcntl.h>
#include <stdio.h>

void redirect_stdout(char *);

main()
{
       printf("Hello world\n");       /*this printf goes to
                                      * standard output*/
       fflush(stdout);
       redirect_stdout("foo");        /*redirect standard output*/
       printf("Hello to you too, foo\n");
                                      /*printf goes to file foo */
       fflush(stdout);
}
void
redirect_stdout(char *filename)
{
        int fd;
        if ((fd = open(filename,O_CREAT|O_WRONLY,0666)) < 0)
                                        /*open a new file */
        {
                perror(filename);
                exit(1);
        }
        close(1);                       /*close old */
                                        *standard output*/
        if (dup(fd) !=1)                /*dup new fd to
                                        *standard input*/
        {
                fprintf(stderr,"Unexpected dup failure\n");
                exit(1);
        }
        close(fd);                       /*close original, new fd,*/
                                         * no longer needed*/
}
```

The value for file descriptor 2 can also be reassigned, but this is rarely done.

Within the file descriptor table, file descriptor numbers are assigned the lowest descriptor number available at the time of a request for a descriptor. However, any value can be assigned within the file descriptor table by using the **dup** subroutine.

File Descriptor Resource Limit

The number of file descriptors that can be allocated to a process is governed by a resource limit. The default value is set in the **/etc/security/limits** file and is typically **2000** (for compatibility with earlier releases). The limit can be changed by the **ulimit** command or the **setrlimit** subroutine. The maximum size is defined by the constant **OPEN_MAX**.

File Creation and Removal

The internal procedures performed by the operating system when creating, opening, or closing files are described in the following sections.

Creating a File

Different subroutines create specific types of files. They are:

Subroutine	Type of File Created
creat	Regular
open	Regular (when the **O_CREAT** flag is set)
mknod	Regular, first-in-first-out (FIFO), or special
mkfifo	Named pipe (FIFO)
pipe	Unnamed pipe
socket	Sockets
mkdir	Directories
symlink	Symbolic link

Creating a Regular File (creat, open, or mknod Subroutines)

You use the **creat** subroutine to create a file according to the values set in the *Pathname* and *Mode* parameters. If the file named in the *Pathname* parameter exists and the process has write permission to the file, the **creat** subroutine truncates the file. Truncation releases all data blocks and sets the file size to 0. You can also create new, regular files using the **open** subroutine with the **O_CREAT** flag.

Files created with the **creat**, **mkfifo**, or **mknod** subroutine take the access permissions set in the *Mode* parameter. Regular files created with the **open** subroutine take their access modes from the **O_CREAT** flag *Mode* parameter. The **umask** subroutine sets a file-mode creation mask (set of access modes) for new files created by processes and returns the previous value of the mask.

The permission bits on a newly created file are a result of the reverse of the **umask** bits ANDed with the file-creation mode bits set by the creating process. When a new file is created by a process, the operating system performs the following actions:

- Determines the permissions of the creating process.
- Retrieves the appropriate **umask** value.
- Reverses the **umask** value.
- Uses the AND operation to combine the permissions of the creating process with the reverse of the **umask** value.

Creating a Special File (mknod or mkfifo Subroutine)

You can use the **mknod** and **mkfifo** subroutines to create new special files. The **mknod** subroutine handles named pipes (FIFO), ordinary, and device files. It creates an i-node for a file identical to that created by the **creat** subroutine. When you use the **mknod** subroutine, the file-type field is set to indicate the type of file being created. If the file is a block or character-type device file, the names of the major and minor devices are written into the i-node.

The **mkfifo** subroutine is an interface for the **mknod** subroutine and is used to create named pipes.

Opening a File

The **open** subroutine is the first step required for a process to access an existing file. The **open** subroutine returns a file descriptor. Reading, writing, seeking, duplicating, setting I/O parameters, determining file status and closing the file all use the file descriptor returned by the **open** call. The **open** subroutine creates entries for a file in the file descriptor table when assigning file descriptors.

The **open** subroutine:

- Checks for appropriate permissions that allow the process access to the file.
- Assigns a entry in the file descriptor table for the open file. The **open** subroutine sets the initial read/write byte offset to 0, the beginning of the file.

The **ioctl** or **ioctlx** subroutines perform control operations on opened special device files.

Closing a File

When a process no longer needs access to the open file, the **close** subroutine removes the entry for the file from the table. If more than one file descriptor references the file table entry for the file, the reference count for the file is decreased by 1, and the close completes. If a file has only 1 reference to it, the file table entry is freed. Attempts by the process to use the disconnected file descriptor result in errors until another **open** subroutine reassigns a value for that file descriptor value. When a process exits, the kernel examines its active user file descriptors and internally closes each one. This ensures that all files close before the process ends.

Working with File I/O

All input and output (I/O) operations use the current file offset information stored in the system file structure (“System File and File Descriptor Tables” on page 125). The current I/O offset designates a byte offset that is constantly tracked for every open file. It is called the current I/O offset because it signals a read or write process where to begin operations in the file. The **open** subroutine resets it to 0. The pointer can be set or changed using the **lseek** subroutine.

Manipulating the Current Offset

Read and write operations can access a file sequentially. This is because the current I/O offset of the file tracks the byte offset of each previous operation. The offset is stored in the system file table.

You can adjust the offset on files that can be randomly accessed, such as regular and special-type files, using the **lseek** subroutine.

lseek Allows a process to position the offset at a designated byte. The **lseek** subroutine positions the pointer at the byte designated by the *Offset* variable. The *Offset* value can be calculated from three places in the file (designated by the value of the *Whence* variable):

absolute offset
Beginning byte of the file

relative offset
Position of the former pointer

end_relative offset
End of the file

The return value for the **lseek** subroutine is the current value of the pointer's position in the file. For example:

```
cur_off= lseek(fd, 0, SEEK_CUR);
```

The **lseek** subroutine is implemented in the file table. All following read and write operations use the new position of the offset as their starting location.

Note: The offset cannot be changed on pipes or socket-type files.

fclear Subroutine that creates an empty space in a file. It sets to zero the number of bytes designated in the *NumberOfBytes* variable beginning at the current offset. The **fclear** subroutine cannot be used if the **O_DEFER** flag was set at the time the file was opened.

Reading a File

The **read** Subroutine that copies a specified number of bytes from an open file to a specified buffer. The copy begins at the point indicated by the current offset. The number of bytes and buffer are specified by the *NBytes* and *Buffer* parameters.

The **read** subroutine:

1. Assures that the *FileDescriptor* parameter is valid and that the process has **read** permissions. The subroutine then gets the file table entry specified by the *FileDescriptor* parameter.
2. Sets a flag in the file to indicate a read operation is in progress. This locks other processes out of the file during the operation.
3. Converts the offset byte value and the value of the *NBytes* variables into a block address.
4. Transfers the contents of the identified block into a storage buffer.
5. Copies the contents of the storage buffer into the area designated by the *Buffer* variable.
6. Updates the current offset according to the number of bytes actually read. Resetting the offset assures that the data is read in sequence by the next read process.
7. Deducts the number of bytes read from the total specified in the *NByte* variable.
8. Loops until the number of bytes to be read is satisfied.
9. Returns the total number of bytes read.

The cycle completes when the file to be read is empty, the number of bytes requested is met, or a reading error is encountered during the process.

Errors can occur while the file is being read from disk or in copying the data to the system file space.

It is advantageous for read requests to start at the beginning of data block boundaries and to be multiples of the data block size. An extra iteration in the read loop can be avoided. If a process reads blocks sequentially, the operating system assumes all subsequent reads will be sequential too.

During the read operation, the i-node is locked. No other processes are allowed to modify the contents of the file while a read is in progress. However the file is unlocked immediately on completion of the read operation. If another process changes the file between two read operations, the resulting data is different, but the integrity of the data structure is maintained.

The following example illustrates how to use the read subroutine to count the number of null bytes in the `foo` file:

```
#include <fcntl.h>
#include <sys/param.h>

main()
```

```
{
        int fd;
        int nbytes;
        int nbytes;
        int nnulls;
        int i;
        char buf[PAGESIZE];      /*A convenient buffer size*/
        nnulls=0;
        if ((fd = open("foo",O_RDONLY)) < 0)
                exit();
        while ((nbytes = read(fd,buf,sizeof(buf))) > 0)
                for (i = 0; i < nbytes; i++)
                        if (buf[i] == '\0';
                                nnulls++;
        printf("%d nulls found\n", nnulls);
}
```

Writing a File

write Subroutine that adds the amount of data specified in the *NBytes* variable from the space designated by the *Buffer* variable to the file described by the *FileDescriptor* variable. It functions similar to the **read** subroutine. The byte offset for the write operation is found in the system file table's current offset.

Sometimes when you write to a file the file does not contain a block corresponding to the byte offset resulting from the write process. When this happens, the **write** subroutine allocates a new block. This new block is added to the i-node information that defines the file. If adding the new block produces an indirect block position (`i_rindirect`), the subroutine allocates more than one block when a file moves from direct to indirect geometry.

During the write operation, the i-node is locked. No other processes are allowed to modify the contents of the file while a write is in progress. However the file is unlocked immediately on completion of the write operation. If another process changes the file between two write operations, the resulting data is different, but the integrity of the data structure is maintained.

The **write** subroutine loops in a way similar to the **read** subroutine, logically writing one block to disk for each iteration. At each iteration, the process either writes an entire block or only a portion of one. If only a portion of a data block is required to accomplish an operation, the **write** subroutine reads the block from disk to avoid overwriting existing information. If an entire block is required, it does not read the block because the entire block is overwritten. The write operation proceeds block by block until the number of bytes designated in the *NBytes* parameter is written.

Delayed Write

You can designate a delayed write process with the **O_DEFER** flag. Then, the data is transferred to disk as a temporary file. The delayed write feature caches the data in case another process reads or writes the data sooner. Delayed write saves extra disk operations. Many programs, such as mail and editors create temporary files in the directory /**tmp** and quickly remove them.

When a file is opened with the deferred update (**O_DEFER**) flag, the data is not written to permanent storage until a process issues an **fsync** subroutine call or a process issues a synchronous **write** to the file (opened with **O_SYNC** flag). The **fsync** subroutine saves all changes in an open file to disk. See the **open** subroutine for a description of the **O_DEFER** and **O_SYNC** flags.

Truncating Files

The **truncate** or **ftruncate** subroutines change the length of regular files. The truncating process must have write permission to the file. The *Length* variable value indicates the size of the file after the truncation operation is complete. All measures are relative to the first byte of the file, not the current offset. If the new length (designated in the *Length* variable) is less than the previous length, the data between the two is

removed. If the new length is greater than the existing length, zeros are added to extend the file size. When truncation is complete, full blocks are returned to the file system, and the file size is updated.

Writing Programs to Use Direct I/O

Beginning in AIX 4.3, an application will be able to use Direct I/O on JFS or JFS2 files. This article is intended to assist programmers in understanding the intricacies involved with writing programs to take advantage of this feature.

Direct I/O vs. Normal Cached I/O

Normally, the JFS or JFS2 caches file pages in kernel memory. When the application does a file read request, if the file page is not in memory, the JFS or JFS2 reads the data from the disk into the file cache, then copies the data from the file cache to the user's buffer. For application writes, the data is merely copied from the user's buffer into the cache. The actual writes to disk are done later.

This type of caching policy can be extremely effective when the cache hit rate is high. It also enables read-ahead and write-behind policies. Lately, it makes file writes to the asynchronous, allowing the application to continue processing instead of waiting for I/O requests to complete.

Direct I/O is an alternative caching policy which causes the file dta to be transferred from the disk to/from the user's buffer. Direct I/O for files is functionally equivalent to raw I/O for devices.

Benefits of Direct I/O

The primary benefit of direct I/O is to reduce CPU utilization for file reads and writes by eliminating the copy from the cache to the user buffer. This can also be a benefit for file data which has a very poor cache hit rate. If the cache hit rate is low, then most read requests have to go to the disk. Direct I/O can also benefit applications which must use synchronous writes since these writes have to go to disk. In both of these cases, CPU usage is reduced since the data copy is eliminated.

A second benefit if direct I/O is that it allows applications to avoid diluting the effectiveness of caching of other files. Any time a file is read or written, that file competes for space in the cache. This may cause other file data to be pushed out of the cache. If the newly cached data has very poor reuse characterisitics, the effectiveness of the cache can be reduced. Direct I/O gives applications the ability to identify files where the normal caching policies are ineffective, thus freeing up more cache space for files where the policies are effective.

Performance Costs of Direct I/O

Although Direct I/O can reduce cpu usage, it typically results in longer wall clock times, especially for relatively small requests. This penalty is caused by the fundamental differences between normal cached I/O and Direct I/O.

Direct I/O Reads

Every Direct I/O read causes a synchronous read from disk; unlike the normal cached I/O policy where read may be satisfied from the cache. This can result in very poor performance if the data was likely to be in memory under the normal caching policy.

Direct I/O also bypasses the normal JFS or JFS2 read-ahead algorithms. These algorithms can be extremely effective for sequential access to files by issuing larger and larger read requests and by overlapping reads of future blocks with application processing.

Applications can compensate for the loss of JFS or JFS2 read-ahead by issuing larger reads requests. At a minimum, Direct I/O readers should issue read requests of at least 128k to match the JFS or JFS2 read-ahead characteristics.

Applications can also simulate JFS or JFS2 read-ahead by issuing asynchronous Direct I/O read-ahead either by use of multiple threads or by using aio_read.

Direct I/O Writes

Every direct I/O write causes a synchronous write to disk; unlike the normal cached I/O policy where the data is merely copied and then written to disk later. This fundamental difference can cause a significant performance penalty for applications which are converted to use Direct I/O.

Conflicting File Access Modes

In order to avoid consistency issues between programs that use Direct I/O and programs that use normal cached I/O, Direct I/O is an exclusive use mode. If there are multiple opens of a file and some of them are direct and others are not, the file will stay in its normal cached access mode. Only when the file is open exclusively by Direct I/O programs will the file be placed in Direct I/O mode.

Similarly, if the file is mapped into virtual memory via the **shmat** or **mmap** system calls, then file will stay in normal cached mode.

The JFS or JFS2 will attempt to move the file into Direct I/O mode any time the last conflicting. non-direct access is eliminated (either by close, munmap, or shmdt). Changing the file from normal mode to Direct I/O mode can be rather expensive since it requires writing all modified pages to disk and removing all the file's pages from memory.

Enabling Applications to use Direct I/O

Applications enable Direct I/O access to a file by passing the O_DIRECT flag to the fcntl.h. This flag is defined in open. Applications must be compiled with _ALL_SOURCE enabled to see the definition of O_DIRECT.

Offset/Length/Address Alignment Requirements of the Target Buffer

In order for Direct I/O to work efficiently, the request should be suitably conditioned. Applications can query the offset, length, and address alignment requirements by using the **finfo** and **ffinfo** subroutines. When the FI_DIOCAP command is used, finfo and ffinfo return information in the diocapbuf structure as described in **sys/finfo.h**. This structure contains the following fields:

dio_offset	Contains the recommended offset alignment for direct I/O writes to files in this file system
dio_max	Contains the recommended maximum write length for Direct I/O writes to files in this system
dio_min	Contains the recommended minimum write length for Direct I/O writes to files in this file system
dio_align	Contains the recommended buffer alignment for Direct I/O writes to files in this file system

Failure to meet these requirements may cause file reads and writes to use the normal cached model. Different file systems may have different requirements.

FS Format	dio_offset	dio_max	dio_min	dio_align
fixed, 4k blk	4k	2m	4k	4k
fragmented	4k	2m	4k	4k
compressed	n/a	n/a	n/a	n/a
big file	128k	2m	128k	4k

Direct I/O Limitations

Direct I/O is not supported for files in a compressed file filesystem. Attempts to open these files with O_DIRECT will be ignored and the files will be accessed with the normal cached I/O methods.

Direct I/O and Data I/O Integrity Completion

Although Direct I/O writes are done synchronously, they do not provide synchronized I/O data integrity completion, as defined by POSIX. Applications which need this feature should use O_DSYNC in addition O_DIRECT. O_DSYNC guarantees that all of the data and enough of the meta-data (eg. indirect blocks) have written to the stable store to be able to retrieve the data after a system crash. O_DIRECT only writes the data; it does not write the meta-data.

Working with Pipes

Pipes are unnamed objects created to allow two processes to communicate. One process reads and the other process writes to the pipe file. This unique type of file is also called a first-in-first-out (FIFO) file. The data blocks of the FIFO are manipulated in a circular queue, maintaining read and write pointers internally to preserve the FIFO order of data. The **PIPE_BUF** system variable, defined in the **limits.h** file, designates the maximum number of bytes guaranteed to be atomic when written to a pipe.

The shell uses unnamed pipes to implement command pipelining. Most unnamed pipes are created by the shell. The | (vertical) symbol represents a pipe between processes. For example:

```
ls | pr
```

the output of the **ls** command is printed to the screen.

Pipes are treated as regular files as far is possible. Normally, the current offset information is stored in the system file table. However, because pipes are shared by processes, the read/write pointers must be specific to the file, not to the process. File table entries are created by the open subroutine and are unique to the open process, not to the file. Processes with access to pipes share the access through common system file table entries.

Using Pipe Subroutines

The **pipe** subroutine creates an interprocess channel and returns two file descriptors. File descriptor 0 is opened for reading. File descriptor 1 is opened for writing. The read operation accesses the data on a FIFO basis. These two file descriptors are used with **read**, **write**, and **close** subroutines.

In the following example, a child process is created and sends its process ID back through a pipe:

```
#include <sys/types.h>
main()
{
        int p[2];
        char buf[80];
        pid_t pid;

        if (pipe(p))
        {
                  perror("pipe failed");
                exit(1)'
        }
        if ((pid=fork()) == 0)
        {
                                    /* in child process */
                close(p[0]);        /*close unused read */
                                    *side of the pipe */
                sprintf(buf,"%d",getpid());
                                    /*construct data */
                                    /*to send */
                write(p[1],buf,strlen(buf)+1);
                        /*write it out, including
                        /*null byte */
                exit(0);
        }
                                     /*in parent process*/
        close(p[1]);                 /*close unused write side                                    /*side of pipe *
        read(p[0],buf,sizeof(buf));  /*read the pipe*/
        printf("Child process said: %s/n", buf);
                                    /*display the result */
        exit(0);
}
```

If a process reads an empty pipe, the process waits until data arrives. If a process writes to a pipe that is too full (**PIPE_BUF**), the process waits until space is available. If the write side of the pipe is closed, a subsequent read operation to the pipe returns an end-of-file.

Two other subroutines that control pipes are the **popen** and **pclose** subroutines.

popen Creates the pipe (using the **pipe** subroutine) then forks to create a copy of the caller. The child process decides whether it is supposed to read or write, closes the other side of the pipe, then calls the shell (using the **execl** subroutine) to run the desired process.

The parent closes the end of the pipe it did not use. These closes are necessary to make end-of-file tests work properly. For example, if a child process intended to read the pipe does not close the write end of the pipe, it will never see the end of file condition on the pipe, because there is one write process potentially active.

The conventional way to associate the pipe descriptor with the standard input of a process is:

```
close(p[1]);
close(0);
dup(p[0]);
close(p[0]);
```

The **close** subroutine disconnects file descriptor 0, the standard input. The **dup** subroutine returns a duplicate of an already open file descriptor. File descriptors are assigned in increasing order and the first available one is returned. The effect of the **dup** subroutine is to copy the file descriptor for the pipe (read side) to file descriptor 0, thus standard input becomes the read side of the pipe. Finally, the previous read side is closed. The process is similar for a child process to write from a parent.

pclose Closes a pipe between the calling program and a shell command to be executed. Use the **pclose** subroutine to close any stream opened with the **popen** subroutine.

The **pclose** subroutine waits for the associated process to end, then closes and returns the exist status of the command. This subroutine is preferable to the close subroutine because **pclose** waits for child processes to finish before closing the pipe. Equally important, when a process creates several children, only a bounded number of unfinished child processes can exist, even if some of them have completed their tasks. Performing the wait allows child processes to complete their tasks.

Synchronous I/O

By default, writes to files in JFS or JFS2 file systems are asynchronous. However, JFS file systems support three types of synchronous I/O. One type is specified by the O_DSYNC open flag. When a file is opened using the O_DSYNC open mode, the write () system call will not return until the file data and all file system meta-data required to retrieve the file data are both written to their permanent storage locations.

Another type of synchronous I/O is specified by the O_SYNC open flag. In addition to items specified by O_DSYNC, O_SYNC specifies that the write () system call will not return until all file attributes relative to the I/O are written to their permanent storage locations, even if the attributes are not required to retrieve the file data.

Before the O_DSYNC open mode existed, AIX applied O_DSYNC semantics to O_SYNC. For binary compatibility reasons, this behavior can never change. If true O_SYNC behavior is required, then both O_DSYNC and O_SYNC open flags must be specified. Exporting the XPG_SUS_ENV=ON environment variable also enables true O_SYNC behavior.

The last type of synchronous I/O is specified by the O_RSYNC open flag, and it simply applies the behaviors associated with O_SYNC or _DSYNC to reads. For files in JFS file systems, only the

combination of O_RSYNC | O_SYNC has meaning. It means that the read system call will not return until the file's access time is written to its permanent storage location.

File Status

File status information resides in the i-node. The **stat** subroutines are used to return information on a file. The **stat** subroutines report file type, file owner, access mode, file size, number of links, i-node number, and file access times. These subroutines write information into a data structure designated by the *Buffer* variable. The process must have search permission for the directories in the path to the designated file.

stat Subroutine that returns the information about files named by the *Path* parameter. If the size of the file cannot be represented in the structure designated by the *Buffer* variable, **stat** will fail with the errno set to EOVERFLOW.

lstat Subroutine that provides information about a symbolic link, and the **stat** subroutine returns information about the file referenced by the link. The **fstat** subroutine returns information from an open file using the file descriptor.

The **statfs**, **fstafs**, and **ustat** subroutines return status information about a file system. The **statfs** subroutine returns information about the file system that contains the file specified by the *Path* parameter.

fstatfs Returns the information about the file system that contains the file associated with the given file descriptor. The structure of the returned information is described in the **/usr/include/sys/statfs.h** file for the **statfs** and **fstatfs** subroutines and in the **ustat.h** file for the **ustat** subroutine.

ustat Returns information about a mounted file system designated by the *Device* variable. This device identifier is for any given file and can be determined by examining the `st_dev` field of the **stat** structure defined in the **/usr/include/sys/stat.h** file. The **ustat** subroutine is superseded by the **statfs** and **fstatfs** subroutines.

utimes and **utime** Also affect file status information. They change the file access and modification time in the i-node.

File Accessibility

Every file is created with an access mode. Each access mode grants read, write, or execute permission to users, the user's group, and all other users.

The access bits on a newly created file are a result of the reverse of the **umask** bits ANDed with the file-creation mode bits set by the creating process. When a new file is created by a process, the operating system performs the following actions:

- Determines the permissions of the creating process
- Retrieves the appropriate **umask** value
- Reverses the **umask** value
- Uses the AND operation to combine the permissions of the creating process with the reverse of the **umask** value

For example, if an existing file has the 027 permissions bits set, the user is not allowed any permissions. Write permission is granted to the group. Read, write, and execute access is set for all others. The **umask** value of the 027 permissions modes would be 750 (the opposite of the original permissions). When 750 is ANDed with 666 (the file creation mode bits set by the system call that created the file), the actual permissions for the file are 640. Another representation of this example is:

```
027 = _ _ _  _ W _  R W X       Existing file access mode
750 = R W X  R _ X  _ _ _       Reverse (umask) of original
                                permissions
666 = R W _  R W _  R W _       File creation access mode
```

```
ANDED TO
750 = R W X  R _ X  _ _ _        The umask value
640 = R W _  R _ _  _ _ _        Resulting file access mode
```

umask Subroutine that sets and gets the value of the file creation mask.

chmod and **fchmod** Subroutines that change file access permissions.

access Subroutine that investigates and reports on the accessibility mode of the file named in the *Pathname* parameter. This subroutine uses the real user ID and the real group ID instead of the effective user and group ID. Using the real user and group IDs allows programs with the set-user-ID and set-group-ID access modes to limit access only to users with proper authorization.

Consider the following example:

```
$ ls -l
total 0
-r-s--x--x      1 root  system   8290 Jun 09 17:07 special
-rw-------      1 root  system   1833 Jun 09 17:07 secrets
$ cat secrets
cat: cannot open secrets
```

In this example, the user does not have access to the file `secrets`. However, when the program `special` is run and the access mode for the program is set-uID `root`, the program can access the file. The program must use the **access** subroutine to prevent subversion of system security.

The **access** subroutine must be used by any set-uID or set-gID program to forestall this type of intrusion.

chown Subroutine resets the ownership field of the i-node for the file and clears the previous owner. The new information is written to the i-node and the process finishes.

The **chmod** subroutine works in similar fashion, but the permission mode flags are changed instead of the file ownership.

Changing file ownership and access modes are actions that affect the i-node, not the data in the file. The owner of the process must have root user authority or own the file to make these changes.

JFS File System Layout

A file system is a set of files, directories, and other structures. File systems maintain information and identify where a file or directory's data is located on the disk. In addition to files and directories, file systems contain a boot block, a superblock, bitmaps, and one or more allocation groups. An allocation group contains disk i-nodes and fragments. Each file system occupies one logical volume.

Boot Block

The boot block occupies the first 4096 bytes of the file system starting at byte offset 0 on the disk. The boot block is available to start the operating system.

Superblock

The superblock is 4096 bytes in size and starts at byte offset 4096 on the disk. The super- block maintains information about the entire file system and includes the following fields:

- Size of the file system
- Number of data blocks in the file system
- A flag indicating the state of the file system

- Allocation group sizes

Allocation Bitmaps

The file system contains two allocation bitmaps:

- The fragment allocation map records the allocation state of each fragment.
- The disk i-node allocation map records the status of each i-node.

Fragments

Many file systems have disk blocks or data blocks. These blocks divide the disk into units of equal size to store the data in a file or directory's logical blocks. The disk block may be further divided into fixed-size allocation units called fragments. Some systems do not allow fragment allocations to span the boundaries of the disk block. In other words, a logical block cannot be allocated fragments from different disk blocks.

The journaled file system (JFS), however, provides a view of the file system as a contiguous series of fragments. JFS fragments are the basic allocation unit and the disk is addressed at the fragment level. Thus, fragment allocations can span the boundaries of what might otherwise be a disk block. The default JFS fragment size is 4096 bytes, although you can specify smaller sizes. In addition to containing data for files and directories, fragments also contain disk addresses and data for indirect blocks. "JFS File Space Allocation" on page 109 explains how the operating system allocates fragments.

Disk I-Nodes

A logical block contains a file or directory's data in units of 4096 bytes. Each logical block is allocated fragments for the storage of its data. Each file and directory has an i-node that contains access information such as file type, access permissions, owner's ID, and number of links to that file. These i-nodes also contain ″addresses″ for finding the location on the disk where the data for a logical block is stored.

Each i-node has an array of numbered sections. Each section contains an address for one of the file or directory's logical blocks. These addresses indicate the starting fragment and the total number of fragments included in a single allocation. For example, a file with a size of 4096 bytes has a single address on the i-node's array. Its 4096 bytes of data are contained in a single logical block. A larger file with a size of 6144 bytes has two addresses. One address contains the first 4096 bytes and a second address contains the remaining 2048 bytes (a partial logical block). If a file has a large number of logical blocks, the i-node does not contain the disk addresses. Instead, the i-node points to an indirect block which contains the additional addresses.

Allocation Groups

The set of fragments making up the file system are divided into one or more fixed-sized units of contiguous fragments. Each unit is an allocation group. The first of these groups begins the file system and contains a reserved area occupying the first 32 x 4096 bytes of the group. The first 4096 bytes of this area hold the boot block and the second 4096 bytes hold the file system superblock.

Each allocation group contains a static number of contiguous disk i-nodes which occupy some of the group's fragments. These fragments are set aside for the i-nodes at file system creation and extension time. For the first allocation group, the disk i-nodes occupy the fragments immediately following the reserved block area. For subsequent groups, the disk i-nodes are found at the start of each group. Disk i-nodes are 128 bytes in size and are identified by a unique disk i-node number or i-number. The i-number maps a disk i-node to its location on the disk or to an i-node within its allocation group.

A file system's allocation groups are described by three sizes:

- The fragment allocation group size and the disk i-node allocation group size are specified as the number of fragments and disk i-nodes that exist in each allocation group.
- The default allocation group size is 8 MB.
- Beginning in AIX 4.2, it can be as large as 64 MB.

These three values are stored in the file system superblock, and they are set at file system creation.

Allocation groups allow the JFS resource allocation policies to use effective methods for achieving good file system I/O performance. These allocation policies try to cluster disk blocks and disk i-nodes for related data to achieve good locality for the disk. Files are often read and written sequentially and files within a directory are often accessed together. Also, these allocation policies try to distribute unrelated data throughout the file system in an attempt to minimize free space fragmentation.

Using File System Subroutines

The most used file system subroutines are:

fscntl	Controls file system control operations
getfsent, **getfsspec**, **getfsfile**, **getfstype**, **setfsent**, or **endfsent**	Obtain information about a file system
lseek	Moves the read-write pointer
mntctl	Returns mount status information
vmount or **mount**	Make a file system ready for use
statfs, **fstsfs**, or **ustat**	Report file system statistics
sync	Updates file systems to disk

Other subroutines are designed for use on virtual file systems (VFS):

getvfsent, **getvfsbytype**, **getvfsbyname**, **getvfsbyflag**, **sevfsent**, or **endvfsent**	Retrieve a VFS entry
umount or **uvmount**	Remove VFS from the file tree

JFS2 File System Layout

A file system is a set of files, directories and other structures. The file systems maintain information and identify where the data is located on the disk for a file or directory. In addition to files and directories a JFS2 file system contains a superblock, allocation maps and one or more allocation groups. An allocation group contains disk inodes and extents. Each file system occupies one logical volume.

Superblock

The superblock is 4096 bytes in size and starts at byte offset 32768 on the disk. The superblock maintains information about the entire file system and includes the following fields:

- Size of the file system
- Number of data blocks in the file system
- A flag indicating the state of the file system
- Allocation group sizes
- File system block size

Allocation Maps

The file system contains two allocation maps:

- The inode allocation map records the location and allocation of all inodes in the file system.

- The block allocation map records the allocation state of each file system block.

Disk I-Nodes

A logical block contains a file or directory's data in units of file system blocks. Each logical block is allocated file system blocks for the storage of its data. Each file and directory has an i-node that contains access information such as file type, access permissions, owner's ID, and number of links to that file. These i-nodes also contain a "B+-tree" for finding the location on the disk where the data for a logical block is stored.

Allocation Groups

Allocation groups divide the space on a file system into chunks. Allocation groups are used for heuristics only. Allocation groups allow JFS2 resource allocation policies to use well known methods for achieving good I/O performance. First, the allocation policies try to cluster disk blocks and disk inodes for related data to achieve good locality for the disk. Files are often read and written sequentially and the files within a directory are often accessed together. Second, the allocation policies try to distribute unrelated data throughout the file system in order to accomodate disk locality.

Alocation groups within a file system are identified by a zero-based allocation group index, the allocation group number.

Allocation Group Sizes

Allocation group sizes must be selected which yield allocation groups that are sufficiently large to provide for contiguous resource allocation over time. Allocation groups are limited to a maximum number of 128 groups. Additionally, the minimum allocation group size is 8192 file system blocks.

Partial Allocation Groups

A file system whose size is not a multiple of the allocation group size will contain a partial allocation group; the last allocation group of the file system is not fully covered by disk blocks. This partial allocation group will be treated as a complete allocation group, except the non-existant disk blocks will be marked as allocated in the block allocation map.

heuristics
: Relating to or using a problem-solving technique in which the most appropriate solution of several found by alternative methods is selected at successive stages of a program for use in the next step of the program.

Using File System Subroutines

The most used file system subroutines are:

fscntl	Controls file system control operations
getfsent, **getfsspec**, **getfsfile**, **getfstype**, **setfsent**, or **endfsent**	Obtain information about a file system
lseek	Moves the read-write pointer
mntctl	Returns mount status information
vmount or **mount**	Make a file system ready for use
statfs, **fstsfs**, or **ustat**	Report file system statistics
sync	Updates file systems to disk

Other subroutines are designed for use on virtual file systems (VFS):

getvfsent, **getvfsbytype**, **getvfsbyname**, **getvfsbyflag**, **sevfsent**, or **endvfsent**	Retrieve a VFS entry
umount or **uvmount**	Remove VFS from the file tree

Creating New File System Types

If it is necessary to create a new type of file system, file system helpers and mount helpers must be created. The following sections provide information about the implementation specifics and execution syntax of file system and mount helpers.

File System Helpers

To enable support of multiple file system types, most file system commands do not contain the code that communicates with individual file systems. Instead, the commands collect parameters, file system names, and other information not specific to one file system type and then pass all this information to a back-end program (the helper).

The back end understands specific information about the relevant file system type and does the detail work of communicating with the file system. Back-end programs used by file system commands are known as file system and mount helpers.

To determine the appropriate file system helper, the front-end command looks for a helper under the directory /sbin/helpers/vfstype/command, where vfstype matches the file system type found in the /etc/vfs file and command matches the name of the command being executed. The flags passed to the front-end command are passed to the file system helper.

There is one file system helper which needs to be provided that does not match a command name. It is called fstype. This helper is used to identify if a specified logical volume contains a file system of the vfstype of the helper. The helper should return 0 if the logical volume does not contain a file system of its type. The helper should return 1 if the logical volume does contain a file system of its type and the file system does not need a separate device for a log. The helper should return 2 if the logical volume does contain a file system of its type and the file system does need a separate device for a log. If the -l lfag is specified, the fstype helper should check for a log of its file system type on the specified logical volume. A return value of 0 indicates the logical volume does not contain a log while a return value of 1 indicates the logical volume does contain a log.

Obsolete File System Helper mechanism

This section describes the obsolete File System helper mechanism which was used on previous versions of AIX. This mechanism is still available but should not be used anymore.

File System Helper Operations

The following table lists the possible operations requested of a helper in the **/usr/include/fshelp.h** file:

Helper Operations	Value
#define FSHOP_NULL	0
#define FSHOP_CHECK	1
#define FSHOP_CHGSIZ	2
#define FSHOP_FINDATA	3
#define FSHOP_FREE	4
#define FSHOP_MAKE	5
#define FSHOP_REBUILD	6
#define FSHOP_STATFS	7
#define FSHOP_STAT	8
#define FSHOP_USAGE	9
#define FSHOP_NAMEI	10
#define FSHOP_DEBUG	11

However, the JFS file system supports only the following operations:

```
Operation Value Corresponding Command

#define FSHOP_CHECK     1    fsck

#define FSHOP_CHGSIZ    2    chfs

#define FSHOP_MAKE      5    mkfs

#define FSHOP_STATFS    7    df

#define FSHOP_NAMEI     10   ff
```

Mount Helpers

The **mount** command is a front-end program that uses a helper to communicate with specific file systems. Helper programs for the **mount** and **umount** (or **unmount**) commands are called mount helpers.

Like other file system-specific commands, the **mount** command collects the parameters and options given at the command line and interprets that information within the context of the file system configuration information found in the **/etc/filesystems** file. Using the information in the **/etc/filesystems** file, the command invokes the appropriate mount helper for the type of file system involved. For example, if the user enters:

```
mount /test
```

the **mount** command checks the **/etc/filesystems** file for the stanza that describes the `/test` file system. From the **/etc/filesystems** file, the **mount** command determines that the `/test` file system is a remote NFS mount from the node named `host1`. The **mount** command also notes any options associated with the mount.

An example **/etc/filesystems** file stanza is:

```
/test:
        dev             = /export
        vfs             = nfs
        nodename        = host1
        options  = ro,fg,hard,intr
```

The file system type (`nfs` in our example) determines which mount helper to invoke. The command compares the file system type to the first fields in the **/etc/vfs** file. The field that matches will have the mount helper as its third field.

Major Control Block Header Files

Chapter 6. Floating-Point Exceptions

This chapter provides information about floating-point exceptions and how your programs can detect and handle them.

The Institute of Electrical and Electronics Engineers (IEEE) defines a standard for floating-point exceptions called the IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754). This standard defines five types of floating-point exception that must be signaled when detected:

- Invalid operation
- Division by zero
- Overflow
- Underflow
- Inexact calculation

When one of these exceptions occurs in a user process, it is signaled either by setting a flag or taking a trap. By default, the system sets a status flag in the Floating-Point Status and Control registers (FPSCR), indicating the exception has occurred. Once the status flags are set by an exception, they are cleared only when the process clears them explicitly or when the process ends. The operating system provides subroutines to query, set, or clear these flags.

The system can also cause the floating-point exception signal (**SIGFPE**) to be raised if a floating-point exception occurs. Because this is not the default behavior, the operating system provides subroutines to change the state of the process so the signal is enabled. When a floating-point exception raises the **SIGFPE** signal, the process terminates and produces a core file if no signal-handler subroutine is present in the process. Otherwise, the process calls the signal-handler subroutine.

Floating-Point Exception Subroutines

Floating-point exception subroutines can be used to:

- Change the execution state of the process
- Enable the signaling of exceptions
- Disable exceptions or clear flags
- Determine which exceptions caused the signal
- Test the exception sticky flags

The following subroutines are provided to accomplish these tasks:

fp_any_xcp or **fp_divbyzero**	Test the exception sticky flags
fp_enable or **fp_enable_all**	Enable the signaling of exceptions
fp_inexact, **fp_invalid_op**, **fp_iop_convert**, **fp_iop_infdinf**, **fp_iop_infmzr**, **fp_iop_infsinf**, **fp_iop_invcmp**, **fp_iop_snan**, **fp_iop_sqrt**, **fp_iop_vxsoft**, **fp_iop_zrdzr**, or **fp_overflow**	Test the exception sticky flags
fp_sh_info	Determines which exceptions caused the signal
fp_sh_set_stat	Disables exceptions or clear flags
fp_trap	Changes the execution state of the process
fp_underflow	Tests the exception sticky flags
sigaction	Installs signal handler

Floating-Point Trap Handler Operation

To generate a trap, a program must change the execution state of the process using the **fp_trap** subroutine and enable the exception to be trapped using the **fp_enable** or **fp_enable_all** subroutine.

Changing the execution state of the program may slow performance because floating-point trapping causes the process to execute in serial mode.

When a floating-point trap occurs, the **SIGFPE** signal is raised. By default, the **SIGFPE** signal causes the process to terminate and produce a core file. To change this behavior, the program must establish a signal handler for this signal. See the **sigaction**, **sigvec**, or **signal** subroutines for more information on signal handlers.

Exceptions: Disabled and Enabled Comparison

Refer to the following lists for an illustration of the differences between the disabled and enabled processing states and the subroutines that are used.

Exceptions-Disabled Model

The following subroutines test exception flags in the disabled processing state:

- **fp_any_xcp**
- **fp_clr_flag**
- **fp_divbyzero**
- **fp_inexact**
- **fp_invalid_op**
- **fp_iop_convert**
- **fp_iop_infdinf**
- **fp_iop_infmzr**
- **fp_iop_infsi**
- **fp_iop_invcmp**
- **fp_iop_snan**
- **fp_iop_sqrt**
- **fp_iop_vxsoft**
- **fp_iop_zrdzr**
- **fp_overflow**
- **fp_underflow**

Exceptions-Enabled Model

The following subroutines function in the enabled processing state:

fp_enable or **fp_enable_all**	Enable the signaling of exceptions
fp_sh_info	Determines which exceptions caused the signal
fp_sh_set_stat	Disables exceptions or clear flags
fp_trap	Changes the execution state of the process
sigaction	Installs signal handler

Imprecise Trapping Modes

Some systems have *imprecise trapping modes*. This means the hardware can detect a floating-point exception and deliver an interrupt, but processing may continue while the signal is delivered. As a result, the instruction address register (IAR) is at a different instruction when the interrupt is delivered.

Imprecise trapping modes cause less performance degradation than *precise trapping mode*. However, some recovery operations are not possible, because the operation that caused the exception cannot be determined or because subsequent instruction may have modified the argument that caused the exception.

To use imprecise exceptions, a signal handler must be able to determine if a trap was precise or imprecise.

Precise Traps
In a precise trap, the instruction address register (IAR) points to the instruction that caused the trap. A program can modify the arguments to the instruction and restart it, or fix the result of the operation and continue with the next instruction. To continue, the IAR must be incremented to point to the next instruction.

Imprecise Traps
In an imprecise trap, the IAR points to an instruction beyond the one that caused the exception. The instruction to which the IAR points has not been started. To continue execution, the signal handler does not increment the IAR.

To eliminate ambiguity, the `trap_mode` field is provided in the **fp_sh_info** structure. This field specifies the trapping mode in effect in the user process when the signal handler was entered. This information can also be determined by examining the Machine Status register (MSR) in the **mstsave** structure.

The **fp_sh_info** subroutine allows a floating-point signal handler to determine if the floating-point exception was precise or imprecise.

> **Note:** Even when precise trapping mode is enabled some floating-point exceptions may be imprecise (such as operations implemented in software). Similarly, in imprecise trapping mode some exceptions may be precise.

When using imprecise exceptions, some parts of your code may require that all floating-point exceptions are reported before proceeding. The **fp_flush_imprecise** subroutine is provided to accomplish this. It is also recommended that the **atexit** subroutine be used to register the **fp_flush_imprecise** subroutine to run at program exit. Running at exit ensures that the program does not exit with unreported imprecise exceptions.

Hardware-Specific Subroutines
Some systems have hardware instructions to compute the square root of a floating-point number and to convert a floating-point number to an integer. Models not having these hardware instructions use software subroutines to do this. Either method can cause a trap if the invalid operation exception is enabled. The software subroutines report, through the **fp_sh_info** subroutine, that an imprecise exception occurred, because the IAR does not point to a single instruction that can be restarted to retry the operation.

Example of a Floating-Point Trap Handler
```
/*
 * This code demonstates a working floating-point exception
 * trap handler. The handler simply identifies which
 * floating-point exceptions caused the trap and return.
 * The handler will return the default signal return
 * mechanism longjmp().
 */
#include <signal.h>
#include <setjmp.h>
#include <fpxcp.h>
#include <fptrap.h>
#include <stdlib.h>
#include <stdio.h>
```

```
#define EXIT_BAD  -1
#define EXIT_GOOD  0

/*
 * Handshaking variable with the signal handler. If zero,
 * then the signal hander returns via  the default signal
 * return mechanism; if non-zero, then the signal handler
 * returns via longjmp.
 */
static int fpsigexit;
#define SIGRETURN_EXIT 0
#define LONGJUMP_EXIT  1

static jmp_buf jump_buffer;      /* jump buffer */
#define JMP_DEFINED 0            /* setjmp rc on initial call */
#define JMP_FPE     2            /* setjmp rc on return from */
                                 /* signal handler */

/*
 * The fp_list structure allows text descriptions
 * of each possible trap type to be tied to the mask
 * that identifies it.
 */

typedef struct
  {
  fpflag_t mask;
  char     *text;
  } fp_list_t;

/* IEEE required trap types */

fp_list_t
trap_list[] =
  {
      { FP_INVALID,      "FP_INVALID"},
      { FP_OVERFLOW,     "FP_OVERFLOW"},
      { FP_UNDERFLOW,    "FP_UNDERFLOW"},
      { FP_DIV_BY_ZERO,  "FP_DIV_BY_ZERO"},
      { FP_INEXACT,      "FP_INEXACT"}
  };

/* INEXACT detail list -- this is an system extension */

fp_list_t
detail_list[] =
  {
      { FP_INV_SNAN,   "FP_INV_SNAN" } ,
      { FP_INV_ISI,    "FP_INV_ISI" } ,
      { FP_INV_IDI,    "FP_INV_IDI" } ,
      { FP_INV_ZDZ,    "FP_INV_ZDZ" } ,
      { FP_INV_IMZ,    "FP_INV_IMZ" } ,
      { FP_INV_CMP,    "FP_INV_CMP" } ,
      { FP_INV_SQRT,   "FP_INV_SQRT" } ,
      { FP_INV_CVI,    "FP_INV_CVI" } ,
      { FP_INV_VXSOFT, "FP_INV_VXSOFT" }
  };

/*
 * the TEST_IT macro is used in main() to raise
 * an exception.
 */

#define TEST_IT(WHAT, RAISE_ARG)        \
  {                                     \
  puts(strcat("testing: ", WHAT));      \
  fp_clr_flag(FP_ALL_XCP);              \
  fp_raise_xcp(RAISE_ARG);              \
  }
```

```
/*
 * NAME: my_div
 *
 * FUNCTION:  Perform floating-point division.
 *
 */
double
my_div(double x, double y)
  {
  return x / y;
  }
/*
 * NAME: sigfpe_handler
 *
 * FUNCTION: A trap handler that is entered when
 *           a floating-point exception occurs. The
 *           function determines what exceptions caused
 *           the trap, prints this to stdout, and returns
 *           to the process which caused the trap.
 *
 * NOTES:    This trap handler can return either via the
 *           default return mechanism or via longjmp().
 *           The global variable fpsigexit determines which.
 *
 *           When entered, all floating-point traps are
 *           disabled.
 *
 *           This sample uses printf(). This should be used
 *           with caution since printf() of a floating-
 *           point number can cause a trap, and then
 *           another printf() of a floating-point number
 *           in the signal handler will corrupt the static
 *           buffer used for the conversion.
 *
 * OUTPUTS:  The type of exception that caused
 *           the trap.
 */
static void
sigfpe_handler(int sig,
               int code,
               struct sigcontext *SCP)
  {
  struct mstsave * state = &SCP->sc_jmpbuf.jmp_context;
  fp_sh_info_t flt_context;     /* structure for fp_sh_info()
                                /* call */
  int i;                        /* loop counter */
  extern int fpsigexit;         /* global handshaking variable */
  extern jmp_buf jump_buffer    /*  */

  /*
   * Determine which floating-point exceptions caused
   * the trap. fp_sh_info() is used to build the floating signal
   * handler info  structure, then the member
   * flt_context.trap can be examined. First the trap type is
   * compared for the IEEE required traps, and if the trap type
   * is an invalid operation, the detail bits are examined.
   */

  fp_sh_info(SCP, &flt_context, FP_SH_INFO_SIZE);
static void
sigfpe_handler(int sig,
               int code,
               struct sigcontext *SCP)
  {
  struct mstsave * state = &SCP->sc_jmpbuf.jmp_context;
  fp_sh_info_t flt_context;     /* structure for fp_sh_info()
```

```
                              /* call */
int i;                        /* loop counter */
extern int fpsigexit;         /* global handshaking variable */
extern jmp_buf jump_buffer;   /*  */

/*
 * Determine which floating-point exceptions caused
 * the trap. fp_sh_info() is used to build the floating signal
 * handler info  structure, then the member
 * flt_context.trap can be examined. First the trap type is
 * compared for the IEEE required traps, and if the trap type
 * is an invalid operation, the detail bits are examined.
 */

fp_sh_info(SCP, &flt_context, FP_SH_INFO_SIZE);
for (i = 0; i < (sizeof(trap_list) / sizeof(fp_list_t)); i++)
    {
    if (flt_context.trap & trap_list[i].mask)
      (void) printf("Trap caused by %s error\n", trap_list[i].text);
    }
if (flt_context.trap & FP_INVALID)
    {
    for (i = 0; i < (sizeof(detail_list) / sizeof(fp_list_t)); i++)
        {
        if (flt_context.trap & detail_list[i].mask)
          (void) printf("Type of invalid op is %s\n", detail_list[i].text);
        }
    }
/* report which trap mode was in effect */
switch (flt_context.trap_mode)
    {
  case FP_TRAP_OFF:
    puts("Trapping Mode is OFF");
    break;

  case FP_TRAP_SYNC:
    puts("Trapping Mode is SYNC");
    break;

  case FP_TRAP_IMP:
    puts("Trapping Mode is IMP");
    break;

  case FP_TRAP_IMP_REC:
    puts("Trapping Mode is IMP_REC");
    break;

  default:
    puts("ERROR:  Invalid trap mode");
    }
if (fpsigexit == LONGJUMP_EXIT)
    {
    /*
     * Return via longjmp. In this instance there is no need to
     * clear any exceptions or disable traps to prevent
     * recurrence of the exception, because on return the
     * process will have the signal handler's floating-point
     * state.
     */
    longjmp(jump_buffer, JMP_FPE);
    }
else
    {
    /*
     * Return via default signal handler return mechanism.
```

```
       * In this case you must take some action to prevent
       * recurrence of the trap, either by clearing the
       * exception bit in the fpscr or by disabling the trap.
       * In this case, clear the exception bit.
       * The fp_sh_set_stat routine is used to clear
       * the exception bit.
       */
      fp_sh_set_stat(SCP, (flt_context.fpscr & ((fpstat_t) ~flt_context.trap)));

      /*
       * Increment the iar of the process that caused the trap,
       * to prevent re-execution of the instruction.
       * The FP_IAR_STAT bit in flt_context.flags indicates if
       * state->iar points to an instruction that has logically
       * started. If this bit is true, state->iar points to
       * an operation that has started and will cause another
       * exception if it runs again. In this case you want to
       * continue execution and ignore the results of that
       * operation, so the iar is advanced to point to the
       * next instruction. If the bit is false, the iar already
       * points to the next instruction that must run.
       */

      if ( flt_context.flags & FP_IAR_STAT )
          {
          puts("Increment IAR");
          state->iar += 4;
          }
      }
  return;
  }

/*
 * NAME: main
 *
 * FUNCTION: Demonstrate the sigfpe_handler trap handler.
 *
 */
int
main(void)
  {
  struct sigaction response;
  struct sigaction old_response;
  extern int fpsigexit;
  extern jmp_buf jump_buffer;
  int jump_rc;
  int trap_mode;
  double arg1, arg2, r;

  /*
   * Set up for floating-point trapping. Do the following:
   *  1.  Clear any existing floating-point exception flags.
   *  2.  Set up a SIGFPE signal handler.
   *  3.  Place the process in synchronous execution mode.
   *  4.  Enable all floating-point traps.
   */
  fp_clr_flag(FP_ALL_XCP);
  (void) sigaction(SIGFPE, NULL, &old_response);
  (void) sigemptyset(&response.sa_mask);
  response.sa_flags = FALSE;
  response.sa_handler = (void (*)(int)) sigfpe_handler;
  (void) sigaction(SIGFPE, &response, NULL);
  fp_enable_all();
```

```
/*
 * Demonstate trap handler return via default signal handler
 * return. The TEST_IT macro will raise the floating-point
 * exception type given in its second argument. Testing
 * is done in this case with precise trapping, because
 * it is supported on all platforms to date.
 */
trap_mode = fp_trap(FP_TRAP_SYNC);
if ((trap_mode == FP_TRAP_ERROR) ||
    (trap_mode == FP_TRAP_UNIMPL))
    {
    printf("ERROR: rc from fp_trap is %d\n",
           trap_mode);
    exit(-1);
    }

(void) printf("Default signal handler return: \n");
fpsigexit = SIGRETURN_EXIT;
TEST_IT("div by zero", FP_DIV_BY_ZERO);
TEST_IT("overflow",    FP_OVERFLOW);
TEST_IT("underflow",   FP_UNDERFLOW);
TEST_IT("inexact",     FP_INEXACT);
TEST_IT("signaling nan",      FP_INV_SNAN);
TEST_IT("INF - INF",          FP_INV_ISI);
TEST_IT("INF / INF",          FP_INV_IDI);
TEST_IT("ZERO / ZERO",        FP_INV_ZDZ);
TEST_IT("INF * ZERO",         FP_INV_IMZ);
TEST_IT("invalid compare",    FP_INV_CMP);
TEST_IT("invalid sqrt",       FP_INV_SQRT);
TEST_IT("invalid coversion",  FP_INV_CVI);
TEST_IT("software request",   FP_INV_VXSOFT);
/*
 * Next, use fp_trap() to determine what the
 * the fastest trapmode available is on
 * this platform.
 */
trap_mode = fp_trap(FP_TRAP_FASTMODE);
switch (trap_mode)
    {
  case FP_TRAP_SYNC:
    puts("Fast mode for this platform is PRECISE");
    break;

  case FP_TRAP_OFF:
    puts("This platform dosn't support trapping");
    break;
  case FP_TRAP_IMP:
    puts("Fast mode for this platform is IMPRECISE");
    break;
  case FP_TRAP_IMP_REC:
    puts("Fast mode for this platform is IMPRECISE RECOVERABLE");
    break;
  default:
    printf("Unexpected return code from fp_trap(FP_TRAP_FASTMODE): %d\n",
           trap_mode);
    exit(-2);
    }
/*
 * if this platform supports imprecise trapping, demonstate this.
 */

trap_mode = fp_trap(FP_TRAP_IMP);
if (trap_mode != FP_TRAP_UNIMPL)
    {
```

```
    puts("Demonsrate imprecise FP execeptions");
    arg1 = 1.2;
    arg2 = 0.0;
    r = my_div(arg1, arg2);
    fp_flush_imprecise();
    }
/* demonstate trap handler return via longjmp().
 */

(void) printf("longjmp return: \n");
fpsigexit = LONGJUMP_EXIT;
jump_rc = setjmp(jump_buffer);

switch (jump_rc)
    {
  case JMP_DEFINED:
    (void) printf("setjmp has been set up; testing ...\n");
    TEST_IT("div by zero", FP_DIV_BY_ZERO);
    break;

  case JMP_FPE:
    (void) printf("back from signal handler\n");
    /*
     * Note that at this point the process has the floating-
     * point status inherited from the trap handler. If the
     * trap hander did not enable trapping (as the example
     * did not) then this process at this point has no traps
     * enabled.  We create a floating-point exception to
     * demonstrate that a trap does not occur, then re-enable
     * traps.
     */
    (void) printf("Creating overflow; should not trap\n");
    TEST_IT("Overflow", FP_OVERFLOW);
    fp_enable_all();
    break;

  default:
    (void) printf("unexpected rc from setjmp: %d\n", jump_rc);
    exit(EXIT_BAD);
    }
exit(EXIT_GOOD);
}
```

Chapter 7. Input and Output Handling

This chapter provides an introduction to programming considerations for input and output handling and the input and output handling (I/O) subroutines.

The input and output (I/O) library subroutines can send data to or from either devices or files. The system treats devices as if they were I/O files. For example, you must also open and close a device just as you do a file.

Some of the subroutines use standard input and standard output as their input and output channels. For most of the subroutines, however, you can specify a different file for the source or destination of the data transfer. For some subroutines, you can use a file pointer to a structure that contains the name of the file; for others, you can use a file descriptor (that is, the positive integer assigned to the file when it is opened).

The I/O subroutines stored in the C Library (**libc.a**) provide stream I/O. To access these stream I/O subroutines, you must include the **stdio.h** file using the following statement:

```
#include <stdio.h>
```

Some of the I/O library subroutines are macros defined in a header file and some are object modules of functions. In many cases, the library contains a macro and a function that do the same type of operation. Consider the following when deciding whether to use the macro or the function:

- You cannot set a breakpoint for a macro using the **dbx** program.
- Macros are usually faster than their equivalent functions because the preprocessor replaces the macros with actual lines of code in the program.
- Macros result in larger object code after being compiled.
- Functions can have side effects to avoid.

The files, commands, and subroutines used in I/O handling provide the following interfaces:

Low-level (“Low-Level I/O Interfaces”)	Basic open and close functions for files and devices.
Stream (“Stream I/O Interfaces” on page 154)	Read and write I/O for pipes and FIFOs.
Terminal (“Terminal I/O Interfaces” on page 155)	Formatted output and buffering.
Asynchronous (“Asynchronous I/O Interfaces” on page 156)	Concurrent I/O and processing.
Input Language (“Creating an Input Language with the lex and yacc Commands” on page 271)	The **lex** and **yacc** commands generate a lexical analyzer and a parser program for interpreting I/O.

Low-Level I/O Interfaces

Low-level I/O interfaces are direct entry points into a kernel, providing functions such as opening files, reading to and writing from files, and closing files.

The **line** command provides the interface that allows one line from standard input to be read and the following subroutines provide other low-level I/O functions:

open, **openx**, or **creat**	Prepare a file, or other path object, for reading and writing by means of an assigned file descriptor

read, **readx**, **readv**, or **readvx**	Read from an open file descriptor
write, **writex**, **writev**, or **writevx**	Write to an open file descriptor
close	Relinquish a file descriptor

The **open** and **creat** subroutines set up entries in three system tables. A file descriptor indexes the first table, which functions as a per process data area that can be accessed by read and write subroutines. Each entry in this table has a pointer to a corresponding entry in the second table.

The second table is a per-system data base, or file table, that allows an open file to be shared among several processes. The entries in this table indicate if the file was open for reading, writing, or as a pipe, and when the file was closed. There is also an offset to indicate where the next read or write will take place and a final pointer to indicates entry to the third table, which contains a copy of the file's i-node.

The file table contains entries for every instance of an **open** or **create** subroutine on the file, but the i-node table contains only one entry for each file.

> **Note:** While processing an **open** or **creat** subroutine for a special file, the system always calls the device's **open** subroutine to allow any special processing (such as rewinding a tape or turning on a data-terminal-ready modem lead). However, the system uses the **close** subroutine only when the last process closes the file (that is, when the i-node table entry is deallocated). This means that a device cannot maintain or depend on a count of its users unless an exclusive-use device (that prevents a device from being reopened before its closed) is implemented.

When a read or write operation takes place, the user's arguments and the file table entry are used to set up the following variables:

- User address of the I/O target area
- Byte-count for the transfer
- Current location in the file

If the file referred to is a character-type special file, the appropriate read or write subroutine is called to transfer data and update the count and current location. Otherwise, the current location is used to calculate a logical block number in the file.

If the file is an ordinary file, the logical block number must be mapped to a physical block number. A block-type special file need not be mapped. The resulting physical block number is used to read or write the appropriate device.

Block device drivers can provide the ability to transfer information directly between the user's core image and the device in blocks as large as the caller requests without using buffers. The method involves setting up a character-type special file corresponding to the raw device and providing read and write subroutines to create a private, non-shared buffer header with the appropriate information. If desired, separate open and close subroutines can be provided, and a special-function subroutine can be called for magnetic tape.

Stream I/O Interfaces

Stream I/O interfaces provide data as a stream of bytes that is not interpreted by the system, which offers more efficient implementation for networking protocols than character I/O processing. There are no record boundaries when reading and writing using stream I/O. For example, a process reading 100 bytes from a pipe cannot tell if the process that wrote the data into the pipe did a single write of 100 bytes, or two writes of 50 bytes, or even if the 100 bytes came from two different processes.

Stream I/Os can be pipes or FIFOs, first in, first out files. FIFOs are similar to pipes because they allow the data to flow only one way (left to right). However, a FIFO can be given a name and can be accessed by unrelated processes, unlike a pipe. FIFOs are sometimes referred to as named pipes. Because it has a

name, a FIFO can be opened using the standard I/O **fopen** subroutine. To open a pipe, you must call the **pipe** subroutine, which returns a file descriptor, and the standard I/O **fdopen** subroutine to associate an open file descriptor with a standard I/O stream.

Stream I/O interfaces are accessed through the following subroutines and macros:

fclose	Closes a stream
feof, **ferror**, **clearerr**, or **fileno**	Check the status of a stream
fflush	Write all currently buffered characters from a stream
fopen, **freopen**, or **fdopen**	Open a stream
fread or **fwrite**	Perform binary input
fseek, **rewind**, **ftell**, **fgetpos**, or **fsetpos**	Reposition the file pointer of a stream
getc, **fgetc**, **getchar**, or **getw**	Get a character or word from an input stream
gets or **fgets**	Get a string from a stream
getwc, **fgetwc**, or **getwchar**	Get a wide character from an input stream
getws or **fgetws**	Get a string from a stream
printf, **fprintf**, **sprintf**, **wsprintf**, **vprintf**, **vfprintf**, **vsprintf**, or **vwsprintf**	Print formatted output
putc, **putchar**, **fputc**, or **putw**	Write a character or a word to a stream
puts or **fputs**	Write a string to a stream
putwc, **putwchar**, or **fputwc**	Write a character or a word to a stream
putws or **fputws**	Write a wide character string to a stream
scanf, **fscanf**, **sscanf**, or **wsscanf**	Convert formatted input
setbuf, **setvbuf**, **setbuffer**, or **setlinebuf**	Assign buffering to a stream
ungetc or **ungetwc**	Push a character back into the input stream

Terminal I/O Interfaces

Terminal I/O interfaces operate between a process and the kernel, providing functions such as buffering and formatted output.

Every terminal and pseudo-terminal has a tty structure that contains the current terminal group ID. This field identifies the process group to receive the signals associated with the terminal.

Terminal I/O interfaces are accessed through the **iostat** command, which monitors I/O system device loading, and the **uprintfd** daemon, which allows kernel messages to be written to the terminal screen.

A daemon opens a terminal device in order to log error messages to the **/dev/tty** or **/dev/console** file. If background writes are not allowed, disassociate the daemon process from the controlling terminal.

Terminal characteristics can be enabled or disabled through the following subroutines:

cfgetospeed, **cfsetospeed**, **cfgetispeed**, or **cfsetispeed**	Get and set input and output baud rates
ioctl	Performs control functions associated with open file descriptors, such as controlling the ability of background processes to produce output on the control terminal
termdef	Queries terminal characteristics
tcdrain	Waits for output to complete
tcflow	Performs flow control functions
tcflush	Discards data from the specified queue
tcgetaattr	Gets terminal state
tcgetpgrp	Gets foreground process group ID
tcsendbreak	Sends a break on an asynchronous serial data line
tcsetattr	Sets terminal state

ttylock, **ttywait**, **ttyunlock**, or **ttylocked**	Control tty locking functions
ttyname or **isatty**	Get the name of a terminal
ttyslot	Finds the slot in the **utmp** file for the current user

Asynchronous I/O Interfaces

Asynchronous I/O subroutines allow a process to start an I/O operation and have the subroutine return immediately after the operation is started or queued. Another subroutine is required to wait for the operation to complete (or return immediately if the operation is already finished). This means that a process can overlap its execution with its I/O or overlap I/O between different devices. Although asynchronous I/O does not significantly improve performance for a process that is reading from a disk file and writing to another disk file, asynchronous I/O provides significant performance improvements for other types of I/O driven programs, such as programs that dump a disk to a magnetic tape or display an image on an image display.

Although not required, a process performing asynchronous I/O can tell the kernel to notify it when a specified descriptor is ready for I/O (also called signal-driven I/O). The kernel notifies the user process with the **SIGIO** signal.

To use asynchronous I/O, a process must perform three steps:

1. Establish a handler for the **SIGIO** signal. This step is only necessary if notification by the signal is requested.
2. Set the process ID or the process group ID to receive the **SIGIO** signals. This step is only necessary if notification by the signal is requested.
3. Enable asynchronous I/O. The system administrator usually determines whether asynchronous I/O is loaded (enabled). Enabling occurs at system startup.

The following asynchronous I/O subroutines are provided:

aio_cancel	Cancels one or more outstanding asynchronous I/O requests
aio_error	Retrieves the error status of an asynchronous I/O request
aio_read	Reads asynchronously from a file descriptor
aio_return	Retrieves the return status of an asynchronous I/O request
aio_suspend	Suspends the calling process until one or more asynchronous I/O requests is completed
aio_write	Writes asynchronously to a file descriptor
lio_listio	Initiates a list of asynchronous I/O requests with a single call
poll or **select**	Check I/O status of multiple file descriptors and message queues

For use with the poll subroutine, the following header files are supplied:

poll.h	Defines the structures and flags used by the **poll** subroutine
aio.h	Defines the structure and flags used by the **aio_read**, **aio_write**, and **aio_suspend** subroutines

Chapter 8. Large Program Support

This chapter provides information about using the large address-space model to accommodate programs requiring data areas that are larger than conventional segmentation can handle.

Note: The discussion in this chapter only applies to 32-bit processes. For information about the default 32-bit address space model and the 64-bit address space model, see “Program Address Space Overview” on page 535 and “System Memory Allocation Using the malloc Subsystem” on page 545 in this book.

The system hardware divides the currently active 32-bit virtual address space into 16 independent segments, each addressed by a separate segment register. The operating system refers to segment 2 (virtual address `0x20000000`) as the process private segment. This segment contains most of the per-process information, including user data, user stack, kernel stack, and user block.

Because the system places user data and the user stack within a single segment, the system limits the maximum amount of stack and data to slightly less than 256MB. This size is adequate for most applications. The kernel stack and u-block are relatively small and of fixed size. However, certain applications require large initialized or uninitialized data areas in the data section of a program. Other large data areas can be created dynamically with the **malloc**, **brk** or **sbrk** subroutine.

Some programs need larger data areas than allowed by the default address-space model. Programs that need the larger data areas can use the large address-space model to request the necessary amount of data space.

Understanding the Large Address-Space Model

The large address-space model enables large data applications while allowing programs that use a smaller space to follow the smaller model. To allow a program to use the large address-space model, you must set the `o_maxdata` field in the XCOFF header of the program to indicate the amount of data needed.

In the large address-space model, the data in the program is laid out beginning in segment 3 when the value is non-zero. (The data is laid out beginning in segment 3, even if the value is smaller than a segment size.) The program consumes as many segments as needed to hold the amount of data indicated by the `o_maxdata` field, up to a maximum of 8 segments. The program can therefore have up to 2 gigabytes of data.

Other aspects of the program address space remain unchanged. The user stack, kernel stack, and u-block continue to reside in segment 2. Also, the data resulting from loading a private copy of a shared library is placed in segment 2. Only program data is placed in segment 3 or higher.

As a result of this organizational scheme, the user stack is still limited by the size of segment 2. (However, the user stack can be relocated into a shared memory segment.) In addition, fewer segments are available for mapped files.

While the size of initialized data in a program can be large, there is still a restriction on the size and placement of text. In the executable file associated with a program, the offset of the end of the text section plus the size of the loader section must be less than 256MB. This is required so that this read-only portion of the executable will fit into segment 1 (the TEXT segment). Because of these restrictions, a program cannot have a very large text section.

Understanding the Very Large Address-Space Model

The very large address-space model enables large data applications in much the same way as the large address-space model. There are several differences between the two address-space models though. To allow a program to use the very large address-space model, you must set the o_maxdata field in the XCOFF header to indicate the amount of data needed and set the F_DSA flag in the file header.

The data in the very large address-space model is laid out beginning in segment 3 when the o_maxdata value is greater than zero. The program is then allowed to use as many segments as needed to hold the amount of data indicated by the o_maxdata field, up to a maximum of 8 segments. In the very large address-space model though, these data segments for the data are created dynamically instead of all at exec time as in the large address-space model.

Using the very large address-space model will change the way in which the segments for a program are managed. A programs data is laid out starting in segment 3, and consumes as many segments as needed for the initial data heap. The remaining segments are available to use for other purposes such as shmat() or mmap(). Once a segment has been allocated for the data heap though, it can no longer be used for any other purposes, even if the size of the heap is reduced.

Use of the very large address-space model will also change the default behavior of system calls such as shmat() and mmap(). The behavior of these system calls in the vary large address-space model will change so that they start placing files in segment 14 and work down instead of starting in segment 3 and working up to segment 14. The system calls can use any of the available segments as long as they have not been allocated for the data heap.

The very large address-space model will allow programs to specify a maxdata value of 0x80000000, the largest currently allowable value, and still use all of the available segments above segment 3 until they are allocated for the data heap. In the large address-space model theses additional segments would have been allocated for the data heap at exec and thus unavailable for other purposes.

Enabling the Large Address-Space Models

The large address space model is used if any nonzero value is given for the **maxdata** keyword. The vary large address-space model is used if any non-zero value is given for the maxdata keyword and the dsa keyword is used also. Use the **-bmaxdata** option only if the program needs very large data areas.

Use the **-bmaxdata** flag with the **ld** command to enable the large address-space model.

For example, to link a program that will have the maximum 8 segments reserved to it, the following command line could be used:

```
cc sample.o -bmaxdata:0x80000000
```

To link a program with the vary large address space model enabled and that will have the maximum 8 segments reserved to it, the following command line could be used:

```
cc sample.o -bmaxdata:0x80000000/dsa
```

The number `0x80000000` is the number of bytes, in hexadecimal format, equal to eight 256MB segments. Although larger numbers can be used, they are ignored because a maximum of 8 segments can be reserved. The value following the **-bmaxdata** flag can also be specified in decimal or octal format.

Using the following shell commands, you can patch large programs to use large data without linking them again:

```
/usr/bin/echo '\0200\0\0\0'|dd of=executable_file_name bs=4
count=1 seek=19 conv=notrunc
```

Note: Use the full name of the **echo** command (`/usr/bin/echo`) to avoid invoking any of the shell **echo** subcommands by mistake. Also, these shell commands will not work for the very large address-space model. You must link the program again to get the very large program support.

The echo string generates the binary value `0x80000000`. This **dd** command seeks to the proper offset in the executable file and modifies the `o_maxdata` field. Do not use the **dd** command on nonexecutable object files, loadable modules, or shared libraries.

Executing Programs with Large Data Areas

When a program attempts to execute a program with large data areas, the system recognizes the requirement for large data and attempts to modify the soft limit on data size to accommodate that requirement. However, if it does not have permission to modify the soft limit, the program ends.

In addition, it is also possible that the data size specified in the `o_maxdata` field may be too small to accommodate the amount of space required for initialized or uninitialized data. In this case, the process ends, and an error is reported.

The attempt is also unsuccessful if the new soft limit is above the hard limit for the process. For example, the login process usually sets the hard limit to infinity. However, if the calling process has modified its hard limit using either the **ulimit** command in the Bourne shell or the **limit** command in the C shell, the newly modified soft limit may be above the hard limit for the process. In this case, the process will be killed during exec processing. In this situation, the only message you receive is `killed`, which informs you that the process was killed.

For more information on the **ulimit** command in the Bourne shell, see Bourne Shell Special Commands in *AIX 5L Version 5.1 System User's Guide: Operating System and Devices*. For more information about the **limit** command in the C shell, see Command Substitution in the C Shell and Filename Substitution in the C Shell in *AIX 5L Version 5.1 System User's Guide: Operating System and Devices*.

After placing the program's initialized and uninitialized data in segments 3 and beyond, the system computes the break value. The break value defines the end of the process's static data and the beginning of its dynamically allocatable data. Using the **malloc**, **brk** or **sbrk** subroutine, the process is free to move the break value toward the end of the segment identified by the `maxdata` field in the **a.out** header file.

For example, if the value specified in the `maxdata` field in the **a.out** header file is `0x80000000`, then the maximum break value is up to the end of segment 10 or `0xaffffff`. The **brk** subroutine extends the break across segment boundaries, but not beyond the point specified in the `maxdata` field.

The majority of subroutines are unaffected by large data programs. The semantics of the **fork** subroutine remain unchanged. Large data programs can run other large or small programs, as well as load and unload other modules.

The **setrlimit** subroutine allows the soft data limit to be set to any value that does not exceed the hard limit. However, because of the inherent limitation of the address space model used by the process, it may not be able to increase its size to the value that is set.

Special Considerations

Programs with large data spaces require a large amount of paging space. For example, if a program with a 2-gigabyte address space tries to access every page in its address space, the system must have 2 gigabytes of paging space. The operating system page-space monitor terminates processes when paging space runs low. Programs with large data spaces are terminated first because they typically consume a large amount of paging space.

Debugging programs with large data is similar to debugging other programs. The **dbx** command can debug these large programs actively or from a core dump. A full core dump should not be performed because programs with large data areas produce large core dumps, which consume large amounts of file-system space.

Some application programs may be written in such a way that they rely on characteristics of the address space model. Programs in which the large address space is enabled use a different address space model than programs without the large address space enabled. This could cause problems for applications which make assumptions about the address space model they are running in. In general, avoid application programs that make assumptions about the address space model.

Chapter 9. Parallel Programming

Parallel programming should be used to get benefits of new multiprocessor systems, while maintaining a full binary compatibility with existing monoprocessor systems. The parallel programming facilities are based on a new concept of the operating system: threads. The following information introduces threads and the associated programming facilities. It also discusses general topics concerning parallel programming.

Related Information

"Chapter 10. Programming on Multiprocessor Systems" on page 193 highlights specific problems when writing programs for symetric multiprocessor systems.

"Chapter 11. Threads Programming Guidelines" on page 215 provides detailed information about programming with the threads library (**libpthreads.a**).

Understanding Threads

A thread is an independent flow of control that operates within the same address space as other independent flows of controls within a process. In previous versions of AIX, and in most of UNIX systems, thread and process characteristics are grouped into a single entity called a process. In other operating systems, threads are sometimes called ″lightweight processes,″ or the meaning of the word ″thread″ is sometimes slightly different.

In the following pages, we will learn the differences between a thread and a process, and see what ″thread″ really means in AIX.

Read the following to learn more about threads in AIX:

Threads and Processes

In traditional single-threaded process systems, a process has a set of properties. In multi-threaded systems, these properties are divided between processes and threads. For more information, see "Thread Properties".

Process Properties

A process in a multi-threaded system is the changeable entity. It must be considered as an execution frame. It has all traditional process attributes, such as:

- Process ID, process group ID, user ID, and group ID
- Environment
- Working directory.

A process also provides a common address space and common system resources:

- File descriptors
- Signal actions
- Shared libraries
- Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).

Thread Properties

A thread is the schedulable entity. It has only those properties that are required to ensure its independent flow of control. These include the following properties:

- Stack
- Scheduling properties (such as policy or priority)

- Set of pending and blocked signals
- Some thread-specific data.

An example of thread-specific data is the error indicator, **errno**. In multi-threaded systems, **errno** is no longer a global variable, but usually a subroutine returning a thread-specific **errno** value. Some other systems may provide other implementations of **errno**.

Threads within a process must not be considered as a group of processes. All threads share the same address space. This means that two pointers having the same value in two threads refer to the same data. Also, if any thread changes one of the shared system resources, all threads within the process are affected. For example, if a thread closes a file, the file is closed for all threads.

The Initial Thread

When a process is created, one thread is automatically created. This thread is called the *initial thread*. It ensures the compatibility between the old processes with a unique implicit thread and the new multi-threaded processes. The initial thread has some special properties, not visible to the programmer, that ensure binary compatibility between the old single-threaded programs and the multi-threaded operating system. It is also the initial thread that executes the **main** routine in multi-threaded programs.

Threads Implementation

A thread is the schedulable entity, which means that the system scheduler handles threads. These threads, known by the system scheduler, are strongly implementation-dependent. To facilitate the writing of portable programs, libraries provide another kind of thread.

Kernel Threads and User Threads

A *kernel thread* is a kernel entity, like processes and interrupt handlers; it is the entity handled by the system scheduler. A kernel thread runs within a process, but can be referenced by any other thread in the system. The programmer has no direct control over these threads, unless writing kernel extensions or device drivers. See *AIX 5L Version 5.1 Kernel Extensions and Device Support Programming Concepts* for more information about kernel programming.

A *user thread* is an entity used by programmers to handle multiple flows of controls within a program. The API for handling user threads is provided by a library, the *threads library*. A user thread only exists within a process; a user thread in process A cannot reference a user thread in process B. The library uses a proprietary interface to handle kernel threads for executing user threads. The user threads API, unlike the kernel threads interface, is part of a portable programming model. Thus, a multi-threaded program developed on an AIX system can easily be ported to other systems.

On other systems, user threads are simply called *threads*, and *lightweight process* refers to kernel threads.

Thread Models and Virtual Processors

User threads are mapped to kernel threads by the threads library. The way this mapping is done is called the *thread model*. There are three possible thread models, corresponding to three different ways to map user threads to kernel threads.

- M:1 model
- 1:1 model
- M:N model.

The mapping of user threads to kernel threads is done using *virtual processors*. A virtual processor (VP) is a library entity that is usually implicit. For a user thread, the virtual processor behaves as a CPU for a kernel thread. In the library, the virtual processor is a kernel thread or a structure bound to a kernel thread.

In the M:1 model all user threads are mapped to one kernel thread; all user threads run on one VP. The mapping is handled by a library scheduler. All user threads programming facilities are completely handled by the library. This model can be used on any system, especially on traditional single-threaded systems.

In the 1:1 model, each user thread is mapped to one kernel thread; each user thread runs on one VP. Most of the user threads programming facilities are directly handled by the kernel threads.

In the M:N model, all user threads are mapped to a pool of kernel threads; all user threads run on a pool of virtual processors. A user thread may be bound to a specific VP, as in the 1:1 model. All unbound user threads share the remaining VPs. This is the most efficient and most complex thread model; the user threads programming facilities are shared between the threads library and the kernel threads.

Contention Scope and Concurrency Level

The *contention scope* of a user thread defines how it is mapped to a kernel thread. There are two possible contention scopes:

- System contention scope, sometimes called global contention scope

  A system contention scope user thread is a user thread that is directly mapped to one kernel thread. All user threads in a 1:1 thread model have system contention scope.
- Process contention scope, sometimes called local contention scope.

  A process contention scope user thread is a user thread that shares a kernel thread with other (process contention scope) user threads in the process. All user threads in a M:1 thread model have process contention scope.

In an M:N thread model, user threads can have either system or process contention scope. Therefore, an M:N thread model is often referred as a *mixed-scope* model.

The *concurrency level* is a property of M:N threads libraries. It defines the number of VPs used to run the process contention scope user threads. This number cannot exceed the number of process contention scope user threads, and is usually dynamically set by the threads library. The system also sets a limit to the number of available kernel threads.

libpthreads.a POSIX Threads Library

AIX provides a threads library, called **libpthreads.a**, based on the POSIX 1003.1c industry standard for a portable user threads API. Any program written for use with a POSIX thread library can easily be ported for use with another POSIX threads library; only the performance and very few subroutines of the threads library are implementation-dependent. For this reason, multi-threaded programs written for this version of AIX will work on any future version of AIX.

To enhance the portability of the threads library, the POSIX standard made the implementation of several programming facilities optional. See “Threads Library Options” on page 261 for more information about checking the POSIX options.

libpthreads_compat.a POSIX Draft 7 Threads Library

AIX provides binary compatibility for existing multi-threads applications that were coded to Draft 7 of the POSIX thread standard. These applications will run without re-linking.

The **libpthreads_compat.a** library is actually provided for program development. AIX 4.3 provides program support for both Draft 7 of the POSIX Thread Standard and Xopen Version 5 Standard, which includes the final POSIX 1003.1c Pthread Standard.

See “Developing Multi-Threaded Programs” on page 173 for more information.

Related Information

Note: **Note:** In this book and the related articles, the word *thread* used alone refers to *user threads*. This also applies to user-mode environment programming references, but not to articles related to kernel programming.

Thread Programming Concepts

The following information provides an overview of the threads library and introduces major programming concepts for multi-threaded programming. Unless otherwise specified, the threads library always operates within a single process.

Basic Operations

Basic thread operations include thread creation “Thread Creation” and termination “Thread Termination”.

Thread Creation

Thread creation differs from process creation in that no parent-child relation exists between threads. All threads, except the *initial thread* automatically created when a process is created, are on the same hierarchical level. A thread does not maintain a list of created threads, nor does it know the thread that created it.

When creating a thread, an entry-point routine and an argument must be specified. Every thread has an entry-point routine with one argument. The same entry-point routine may be used by several threads. See “Creating Threads” on page 216 for more information about thread creation.

Thread Termination

Threads can terminate themselves by either returning from their entry-point routine or calling a library subroutine. Threads can also terminate other threads, using a mechanism called *cancellation*. Any thread can request the cancellation of another thread. Each thread controls whether it may be canceled or not. Cleanup handlers may also be registered to perform operations when a cancellation request is acted upon. See “Terminating Threads” on page 219 for more information about thread termination.

Synchronization

Threads need to synchronize their activities to effectively interact. This includes:

- Implicit communication through the modification of shared data
- Explicit communication by informing each other of events that have occurred.

The threads library provides three synchronization mechanisms: mutexes, condition variables, and joins. These are primitive but powerful mechanisms, which can be used to build more complex mechanisms.

Mutexes and Race Conditions

Mutual exclusion locks (mutexes) can prevent data inconsistencies due to race conditions. A race condition often occurs when two or more threads need to perform operations on the same memory area, but the results of computations depends on the order in which these operations are performed.

Consider, for example, a single counter, X, that is incremented by two threads, A and B. If X is originally 1, then by the time threads A and B increment the counter, X should be 3. Both threads are independent entities and have no synchronization between them. Although the C statement `X++` looks simple enough to be atomic, the generated assembly code may not be, as shown in the following pseudo-assembler code:

```
move    X, REG
inc     REG
move    REG, X
```

If both threads are executed concurrently on two CPUs, or if the scheduling makes the threads alternatively execute on each instruction, the following steps may occur:

1. Thread A executes the first instruction and puts X, which is 1, into the thread A register. Then thread B executes and puts X, which is 1, into the thread B register. The following example illustrates the resulting registers and the contents of memory X.

   ```
   Thread A Register = 1
   Thread B Register = 1
   Memory X          = 1
   ```

2. Next, thread A executes the second instruction and increments the content of its register to 2. Then thread B increments its register to 2. Nothing is moved to memory X, so memory X stays the same. The following example illustrates the resulting registers and the contents of memory X.

   ```
   Thread A Register = 2
   Thread B Register = 2
   Memory X          = 1
   ```

3. Last, thread A moves the content of its register, which is now 2, into memory X. Then thread B moves the content of its register, which is also 2, into memory X, overwriting thread A's value. The following example illustrates the resulting registers and the contents of memory X.

   ```
   Thread A Register = 2
   Thread B Register = 2
   Memory X          = 2
   ```

Note that in most cases thread A and thread B will execute the three instructions one after the other, and the result would be 3, as expected. Race conditions are usually difficult to discover, because they occur intermittently.

To avoid this race condition, each thread should lock the data before accessing the counter and updating memory X. For example, if thread A takes a lock and updates the counter, it leaves memory X with a value of 2. Once thread A releases the lock, thread B takes the lock and updates the counter, taking 2 as its initial value for X and incrementing it to 3, the expected result.

See "Using Mutexes" on page 227 for more information about mutexes.

Waiting for Threads

Condition variables allow threads to block until some event or condition has occurred. Boolean predicates indicate whether the program has satisfied a condition variable. The complexity of a condition variable predicate is defined by the programmer. A condition can be signaled by any thread to either one or all waiting threads. See "Using Condition Variables" on page 231 to get more information.

When a thread is terminated, its storage may not be reclaimed, depending on an attribute of the thread. Such threads can be joined by other threads and return information to them. A thread that wants to join another thread is blocked until the target thread terminates. This joint mechanism is a specific case of condition-variable usage, the condition is the thread termination. See "Joining Threads" on page 236 for more information about joins.

Scheduling

The threads library allows the programmer to control the execution scheduling of the threads. The control is performed in different ways:

- By setting scheduling attributes when creating a thread
- By dynamically changing the scheduling attributes of a created thread
- By defining the effect of a mutex on the thread's scheduling when creating a mutex

- By dynamically changing the scheduling of a thread during synchronization operations.

The two last types of controls are known as *synchronization scheduling*.

Scheduling Parameters

A thread has three scheduling parameters:

Scope	The contention scope of a thread is defined by the thread model used in the threads library.
Policy	The scheduling policy of a thread defines how the scheduler treats the thread once it gains control of the CPU.
Priority	The scheduling priority of a thread defines the relative importance of the work being done by each thread.

The scheduling parameters can be set before the thread's creation or during the thread's execution. In general, controlling the scheduling parameters of threads is important only for threads that are compute-intensive. Thus the threads library provides default values that are sufficient for most cases. See "Threads Scheduling" on page 240 for more information about controlling the scheduling parameters of threads.

Synchronization Scheduling

Synchronization scheduling is a complex topic. Some implementations of the threads library do not provide this facility.

Synchronization scheduling defines how the execution scheduling, especially the priority, of a thread is modified by holding a mutex. This allows custom-defined behavior and avoids priority inversions. It is useful when using complex locking schemes. See "Synchronization Scheduling" on page 243 for more information.

Other Facilities

The threads library provides other useful facilities to help programmers implement powerful functions. It also manages the interactions between threads and processes.

Advanced Facilities

The threads library provides an API for handling synchronization and scheduling of threads. It also provides facilities for the following purposes:

- "One-Time Initializations" on page 246 allow dynamic package initializations.
- "Thread-Specific Data" on page 247 allows each thread to maintain its own private data.
- "Advanced Attributes" on page 250 allow control of the size and the address of the thread's stack.

Threads-Processes Interactions

Threads and processes interact when handling specific actions:

- "Signal Management" on page 256 are shared between the process and its threads.
- "Process Duplication and Termination" on page 259 imply thread creation and termination.

Threads Library API

This section provides some general comments about the threads library API. The following information is not required for writing multi-threaded programs, but may help the programmer understand the threads library API.

Object-Oriented Interface

The threads library API provides an object-oriented interface. The programmer manipulates opaque objects using pointers or other universal identifiers. This ensures the portability of multi-threaded programs

between systems that implement the threads library. It also allows implementation changes between two releases of AIX that necessitate only programs to be re-compiled. Although some definitions of data types may be found in the threads library header file (**pthread.h**), programs should not rely on these implementation-dependent definitions to directly handle the contents of structures. The regular threads library subroutines must always be used to manipulate the objects.

The threads library essentially uses three kinds of objects (opaque data types): threads, mutexes, and condition variables. These objects have attributes which specify the object properties. When creating an object, the attributes must be specified. In the threads library, these creation attributes are themselves objects, called *attributes objects*.

Therefore, there are three pairs of objects manipulated by the threads library:

- Threads and thread attributes objects
- Mutexes and mutex attributes objects
- Condition variables and condition attributes objects.

Creating an object requires the creation of an attributes object. An attributes object is created with attributes having default values. Attributes can then be individually modified using subroutines. This ensures that a multi-threaded program will not be affected by the introduction of new attributes or changes in the implementation of an attribute. An attributes object can thus be used to create one or several objects, and then destroyed without affecting objects created with the attributes object.

Using an attributes object also allows the use of object classes. One attributes object may be defined for each object class. Creating an instance of an object class would be done by creating the object using the class attributes object.

Naming Convention

The identifiers used by the threads library follow a strict naming convention. All identifiers of the threads library begin with **pthread_**. User programs should not use this prefix for private identifiers. This prefix is followed by a component name. The following components are defined in the threads library:

pthread_	Threads themselves and miscellaneous subroutines
pthread_attr	Thread attributes objects
pthread_cond	Condition variables
pthread_condattr	Condition attributes objects
pthread_key	Thread-specific data keys
pthread_mutex	Mutexes
pthread_mutexattr	Mutex attributes objects.

Data types identifiers end with **_t**. Subroutines and macros end with an _ (underscore), followed by a name identifying the action performed by the subroutine or the macro. For example, **pthread_attr_init** is a threads library identifier (**pthread_**) concerning thread attributes objects (**attr**) and is an initialization subroutine (**_init**).

Explicit macro identifiers are in uppercase letters. Some subroutines may, however, be implemented as macros, although their names are in lowercase letters.

Related Files

The following AIX files provide the implementation of pthreads:

/usr/include/pthread.h	C/C++ header with most pthread definitions.
/usr/include/sched.h	C/C++ header with some scheduling definitions.
/usr/include/unistd.h	C/C++ header with **pthread_atfork()** definition.
/usr/include/sys/limits.h	C/C++ header with some pthread definitions.
/usr/include/sys/pthdebug.h	C/C++ header with most pthread debug definitions.

/usr/include/sys/sched.h	C/C++ header with some scheduling definitions.
/usr/include/sys/signal.h	C/C++ header with **pthread_kill()** and **pthread_sigmask()** definitions.
/usr/include/sys/types.h	C/C++ header with some pthread definitions.
/usr/lib/libpthreads.a	32-bit/64-bit library providing UNIX98 and POSIX 1003.1c pthreads.
/usr/lib/libpthreads_compat.a	32-bit only library providing POSIX 1003.1c Draft 7 pthreads.
/usr/lib/profiled/libpthreads.a	Profiled 32-bit/64-bit library providing UNIX98 and POSIX 1003.1c pthreads.
/usr/lib/profiled/libpthreads_compat.a	Profiled 32-bit only library providing POSIX 1003.1c Draft 7 pthreads.

Writing Reentrant and Thread-Safe Code

In single-threaded processes there is only one flow of control. The code executed by these processes thus need not to be reentrant or thread-safe. In multi-threaded programs, the same functions and the same resources may be accessed concurrently by several flows of control. To protect resource integrity, code written for multi-threaded programs must be reentrant and thread-safe.

This section provides information for writing reentrant and thread-safe programs. It does not cover the topic of writing thread-efficient programs. Thread-efficient programs are efficiently parallelized programs. This can only be done during the design of the program. Existing single-threaded programs can be made thread-efficient, but this requires that they be completely redesigned and rewritten.

Understanding Reentrance and Thread-Safety

Reentrance and thread-safety are both related to the way functions handle resources. Reentrance and thread-safety are separate concepts: a function can be either reentrant, thread-safe, both, or neither.

Reentrance

A reentrant function does not hold static data over successive calls, nor does it return a pointer to static data. All data is provided by the caller of the function. A reentrant function must not call non-reentrant functions.

A non-reentrant function can often, but not always, be identified by its external interface and its usage. For example, the **strtok** subroutine is not reentrant, because it holds the string to be broken into tokens. The **ctime** subroutine is also not reentrant; it returns a pointer to static data that is overwritten by each call.

Thread-Safety

A thread-safe function protects shared resources from concurrent access by locks. Thread-safety concerns only the implementation of a function and does not affect its external interface.

In C, local variables are dynamically allocated on the stack. Therefore, any function that does not use static data or other shared resources is trivially thread-safe. For example, the following function is thread-safe:

```
/* thread-safe function */
int diff(int x, int y)
{
        int delta;

        delta = y - x;
        if (delta < 0)
                delta = -delta;

        return delta;
}
```

The use of global data is thread-unsafe. It should be maintained per thread or encapsulated, so that its access can be serialized. A thread may read an error code corresponding to an error caused by another thread. In AIX, each thread has its own **errno** value.

Making a Function Reentrant

In most cases, non-reentrant functions must be replaced by functions with a modified interface to be reentrant. Non-reentrant functions cannot be used by multiple threads. Furthermore, it may be impossible to make a non-reentrant function thread-safe.

Returning Data

Many non-reentrant functions return a pointer to static data. This can be avoided in two ways:

- Returning dynamically allocated data. In this case, it will be the caller's responsibility to free the storage. The benefit is that the interface does not need to be modified. However, backward compatibility is not ensured; existing single-threaded programs using the modified functions without changes would not free the storage, leading to memory leaks.
- Using caller-provided storage. This method is recommended, although the interface needs to be modified.

For example, a **strtoupper** function, converting a string to uppercase, could be implemented as in the following code fragment:

```
/* non-reentrant function */
char *strtoupper(char *string)
{
        static char buffer[MAX_STRING_SIZE];
        int index;

        for (index = 0; string[index]; index++)
                buffer[index] = toupper(string[index]);
        buffer[index] = 0

        return buffer;
}
```

This function is not reentrant (nor thread-safe). Using the first method to make the function reentrant, the function would be similar to the following code fragment:

```
/* reentrant function (a poor solution) */
char *strtoupper(char *string)
{
        char *buffer;
        int index;

        /* error-checking should be performed! */
        buffer = malloc(MAX_STRING_SIZE);

        for (index = 0; string[index]; index++)
                buffer[index] = toupper(string[index]);
        buffer[index] = 0

        return buffer;
}
```

A better solution consists of modifying the interface. The caller must provide the storage for both input and output strings, as in the following code fragment:

```
/* reentrant function (a better solution) */
char *strtoupper_r(char *in_str, char *out_str)
{
        int index;

        for (index = 0; in_str[index]; index++)
        out_str[index] = toupper(in_str[index]);
```

```
        out_str[index] = 0

        return out_str;
}
```

The non-reentrant standard C library subroutines were made reentrant using the second method. This is discussed in “Reentrant and Thread-Safe Libraries” on page 172 .

Keeping Data over Successive Calls

No data should be kept over successive calls, because different threads may successively call the function. If a function needs to maintain some data over successive calls, such as a working buffer or a pointer, this data should be provided by the caller.

Consider the following example. A function returns the successive lowercase characters of a string. The string is provided only on the first call, as with the **strtok** subroutine. The function returns 0 when it reaches the end of the string. The function could be implemented as in the following code fragment:

```
/* non-reentrant function */
char lowercase_c(char *string)
{
        static char *buffer;
        static int index;
        char c = 0;

        /* stores the string on first call */
        if (string != NULL) {
                buffer = string;
                index = 0;
        }

        /* searches a lowercase character */
        for (; c = buffer[index]; index++) {
                if (islower(c)) {
                        index++;
                        break;
                }
        }
        return c;
}
```

This function is not reentrant. To make it reentrant, the static data, the **index** variable, needs to be maintained by the caller. The reentrant version of the function could be implemented as in the following code fragment:

```
/* reentrant function */
char reentrant_lowercase_c(char *string, int *p_index)
{
        char c = 0;

        /* no initialization - the caller should have done it */

        /* searches a lowercase character */
        for (; c = string[*p_index]; (*p_index)++) {
                if (islower(c)) {
                        (*p_index)++;
                        break;
                  }
        }
        return c;
}
```

The interface of the function changed and so did its usage. The caller must provide the string on each call and must initialize the index to 0 before the first call, as in the following code fragment:

```
char *my_string;
char my_char;
int my_index;
...
my_index = 0;
while (my_char = reentrant_lowercase_c(my_string, &my_index)) {
        ...
}
```

Making a Function Thread-Safe

In multi-threaded programs, all functions called by multiple threads must be thread-safe. However, there is a workaround for using thread unsafe subroutines in multi-threaded programs. Note also that non-reentrant functions usually are thread-unsafe, but making them reentrant often makes them thread-safe, too.

Locking Shared Resources

Functions that use static data or any other shared resources, such as files or terminals, must serialize the access to these resources by locks in order to be thread-safe. For example, the following function is thread-unsafe:

```
/* thread-unsafe function */
int increment_counter()
{
        static int counter = 0;

        counter++;
        return counter;
}
```

To be thread-safe, the static variable **counter** needs to be protected by a static lock, as in the following (pseudo-code) example:

```
/* pseudo-code thread-safe function */
int increment_counter();
{
        static int counter = 0;
        static lock_type counter_lock = LOCK_INITIALIZER;

        lock(counter_lock);
        counter++;
        unlock(counter_lock);
        return counter;
}
```

In a multi-threaded application program using the threads library, mutexes should be used for serializing shared resources. Independent libraries may need to work outside the context of threads and, thus, use other kinds of locks.

A Workaround for Thread-Unsafe Functions

It is possible to use thread-unsafe functions called by multiple threads using a workaround. This may be useful, especially when using a thread-unsafe library in a multi-threaded program, for testing or while waiting for a thread-safe version of the library to be available. The workaround leads to some overhead, because it consists of serializing the entire function or even a group of functions.

- Use a global lock for the library, and lock it each time you use the library (calling a library routine or using a library global variable), as in the following pseudo-code fragments:

  ```
  /* this is pseudo-code! */

  lock(library_lock);
  library_call();
  unlock(library_lock);

  lock(library_lock);
  x = library_var;
  unlock(library_lock);
  ```

This solution can create performance bottlenecks because only one thread can access any part of the library at any given time. The solution is acceptable only if the library is seldom accessed, or as an initial, quickly implemented workaround.

- Use a lock for each library component (routine or global variable) or group of components, as in the following pseudo-code fragments:

```
/* this is pseudo-code! */

lock(library_moduleA_lock);
library_moduleA_call();
unlock(library_moduleA_lock);

lock(library_moduleB_lock);
x = library_moduleB_var;
unlock(library_moduleB_lock);
```

This solution is somewhat more complicated to implement than the first one, but it can improve performance.

Because this workaround should only be used in application programs and not in libraries, mutexes can be used for locking the library.

Reentrant and Thread-Safe Libraries

Reentrant and thread-safe libraries are useful in a wide range of parallel (and asynchronous) programming environments, not just within threads. Thus it is a good programming practice to always use and write reentrant and thread-safe functions.

Using Libraries

Several libraries shipped with the AIX Base Operating System are thread-safe. In the current version of AIX, the following libraries are thread-safe:

- Standard C library (**libc.a**)
- Berkeley compatibility library (**libbsd.a**).

Some of the standard C subroutines are non-reentrant, such as the **ctime** and **strtok** subroutines. The reentrant version of the subroutines have the name of the original subroutine with a suffix **_r** (underscore r).

When writing multi-threaded programs, the reentrant versions of subroutines should be used instead of the original version. For example, the following code fragment:

```
token[0] = strtok(string, separators);
i = 0;
do {
        i++;
        token[i] = strtok(NULL, separators);
} while (token[i] != NULL);
```

should be replaced in a multi-threaded program by the following code fragment:

```
char *pointer;
...
token[0] = strtok_r(string, separators, &pointer);
i = 0;
do {
        i++;
        token[i] = strtok_r(NULL, separators, &pointer);
} while (token[i] != NULL);
```

Thread-unsafe libraries may be used by only one thread in a program. The uniqueness of the thread using the library must be ensured by the programmer; otherwise, the program will have unexpected behavior, or may even crash.

Converting Libraries

This information highlights the main steps in converting an existing library to a reentrant and thread-safe library. It applies only to C language libraries.

- Identifying exported global variables. Those variables are usually defined in a header file with the **export** keyword.

  Exported global variables should be encapsulated. The variable should be made private (defined with the **static** keyword in the library source code). Access (read and write) subroutines should be created.
- Identifying static variables and other shared resources. Static variables are usually defined with the **static** keyword.

  Locks should be associated with any shared resource. The granularity of the locking, thus choosing the number of locks, impacts the performance of the library. To initialize the locks, the one-time initialization ("One-Time Initializations" on page 246) facility may be used. For more information, see "One-Time Initializations" on page 246
- Identifying non-reentrant functions and making them reentrant. See "Making a Function Reentrant" on page 169
- Identifying thread-unsafe functions and making them thread-safe. See "Making a Function Thread-Safe" on page 171

Developing Multi-Threaded Programs

Developing multi-threaded programs is not much more complicated than developing programs with multiple processes. See "Chapter 11. Threads Programming Guidelines" on page 215 for detailed information about using the threads library. Developing programs also implies compiling and debugging the code.

Compiling a Multi-Threaded Program

This section explains how to generate a multi-threaded program. It describes:

- The required "Header File"
- "Compiler Invocation" to generate multi-threaded programs.

Header File

All subroutine prototypes, macros, and other definitions for using the threads library are in one header file, **pthread.h**, located in the **/usr/include** directory.

The **pthread.h** header file must be the first included file of each source file using the threads library, because it defines some important macros that affect other header files. Having the **pthread.h** header file as the first included file ensures the usage of thread-safe subroutines. The following global symbols are defined in the **pthread.h** file:

_POSIX_REENTRANT_FUNCTIONS	Specifies that all functions should be reentrant. Several header files use this symbol to define supplementary reentrant subroutines, such as the **localtime_r** subroutine.
_POSIX_THREADS	Denotes the POSIX threads API. This symbol is used to check if the POSIX threads API is available. Macros or subroutines may be defined in different ways, depending on whether the POSIX or some other threads API is used.

The **pthread.h** file also redefines the **errno** global variable as a function returning a thread-specific **errno** value. The **errno** identifier is, therefore, no longer an l-value in a multi-threaded program.

Compiler Invocation

When compiling a multi-threaded program, you should invoke the C compiler using one of the following commands:

xlc_r Invokes the compiler with default language level of **ansi**.
cc_r Invokes the compiler with default language level of **extended**.

These commands ensure that the adequate options and libraries are used to be compliant with the X/Open Version 5 Standard. The POSIX Threads Specification 1003.1c is a subset of the X/Open Specification.

The following libraries are automatically linked with your program when using these commands:

libpthreads.a Threads library.
libc.a Standard C library

For example, the following command compiles the **foo.c** multi-threaded C source file and produces the **foo** executable file:

```
cc_r -o foo foo.c
```

Compiler Invocation for Draft 7 of POSIX 1003.1c

AIX provides source code compatibility for Draft 7 applications. It is recommended that developers port their threaded application to the latest standard, which is covered by the compiler directions provided above.

When compiling a multi-threaded program for Draft 7 support of threads, you should invoke the C compiler using one of the following commands:

xlc_r7 Invokes the compiler with default language level of **ansi**.
cc_r7 Invokes the compiler with default language level of **extended**.

The following libraries are automatically linked with your program when using these commands:

libpthreads_compat.a Draft 7 Compatibility Threads library.
libpthreads.a Threads library.
libc.a Standard C library.

Source code compatibility has been achieved through the use of the compiler directive **_AIX_PTHREADS_D7**. It is also necessary to link the libraries in the following order: **libpthreads_compat.a**, **libpthreads.a**, and **libc.a**. Most users do not need to know this information, since the commands listed above provide the necessary options. These options are provided for those that don't have the latest AIX compiler.

Porting Draft 7 applications to the X/Open Version 5 Standard

There are very few differences between Draft 7 and the final standard.

There are some minor **errno** differences. The most prevalent is the use of **ESRCH** to denote the specified pthread could not be found. Draft 7 frequently returned **EINVAL** for this failure.

Pthreads are joinable by default. This is a significant change since it can result in a memory leak if ignored. See "Creating Threads" on page 216 for more information about thread creation.

Pthreads have process scheduling scope by default. See "Threads Scheduling" on page 240 for more information about scheduling.

The subroutine **pthread_yield** has been replaced by **sched_yield**.

The various scheduling policies associated with the mutex locks are slightly different.

Memory Requirements of a Multi-Threaded Program

AIX supports up to 32768 threads in a single process. Each individual pthread requires some amount of process address space so the actual maximum number of pthreads a process can have depends on the memory model and the use of process address space for other purposes. The amount of memory a pthread needs includes the stack size and the guard region size plus some amount for internal use. The user can control the size of the stack with pthread_attr_setstacksize() and the size of the guard region with pthread_attr_setguardsize(). The following table points out the maximum number of pthreads which could be created in a 32-bit process using a simple program which does nothing other than create pthreads in a loop using the NULL pthread attribute. In a real program the actual numbers will depend on other memory usage in the program. For a 64-bit process the ulimit controls how many threads can be created therefore the big data model is not necessary and in fact can decrease the maximum number of threads.

32-bit Process:

Data Model	-bmaxdata:	Maximum Pthreads
Small Data	n/a	1084
Big Data	0x10000000	2169
Big Data	0x20000000	4340
Big Data	0x30000000	6510
Big Data	0x40000000	8681
Big Data	0x50000000	10852
Big Data	0x60000000	13022
Big Data	0x70000000	15193
Big Data	0x80000000	17364

Debugging a Multi-Threaded Program

This section provides an introduction to debugging multi-threaded programs.

Using dbx

Application programmers can use the **dbx** program to perform debugging. Several new subcommands are available for displaying thread-related objects: **attribute**, **condition**, **mutex**, and **thread**.

Using the Kernel Debug Program

Kernel programmers can use the kernel debug program to perform debugging on kernel extensions and device drivers. The kernel debug program provides no access to user threads but handles kernel threads.

Several new commands have been added to support multiple kernel threads and processors: **cpu**, **ppd**, **thread**, and **uthread**. These commands respectively change the current processor, display per-processor data structures, display thread table entries, and display the **uthread** structure of a thread.

Core File Requirements of a Multi-Threaded Program

By default processes do not generate a full core file. Before AIX 4.3 this meant only the stack for the thread causing the core dump was written to the core file. Before AIX 4.3.2 this meant the part of the process address space made up of shared memory region was not written to the core file. If an application needs to debug data in shared memory regions, particular thread stacks it will be necessary to generate a full core dump. To generate full core file information the following command must be run as root:

```
chdev -l sys0 -a fullcore=true
```

Each individual pthread adds to the size of the generated core file. The amount of core file space a pthread needs includes the stack size which the user can control with pthread_attr_setstacksize(). For pthreads created with the NULL pthread attribute each pthread in a 32-bit process adds 128KB to the size of the core file and each pthread in a 64-bit process ads 256KB to the size of the core file.

Developing Multi-Threaded Program which examines and modifies pthread library objects

The **pthread debug library (libpthdebug.a)** provides a set of functions which allow application developers with the capability to examine and modify **pthread library** objects.

This library can be used for both 32-bit applications and 64-bit applications. This library is thread safe. The **pthread debug library** contains a 32-bit shared object and a 64-bit shared object.

The **pthread debug library** provides applicaitons access to the **pthread library** information. This includes information on pthreads, pthread attributes, mutexes, mutex attributes, condition variables, condition variable attributes, read/write locks, read/write lock attributes, and information about the state of the pthread library.

Note: All data (addresses, registers) returned by this library will be in 64-bit format both for 64-bit and 32-bit application. It is the applications responsibility to convert these values into 32-bit format for 32-bit applications. When debugging a 32-bit application the top half of addresses and registers will be ignored.

Note: The **pthread debug library** does not report mutexes, mutexattrs, conds, condattrs, rwlocks, rwlockattrs that have the pshared value of **PTHREAD_PROCESS_SHARED**.

Initialization

The application must initialize a **pthread debug library** session for each pthreaded process. The **pthdb_sessison_init()** function must be called from each pthreaded process after the process has been loaded. The **pthread debug library** supports one session for a single process. The application must assign a unique user identifier and pass it to the **pthdb_session_init()** function which in turn will assign a unique session identifier which must be passed as the first parameter to all other **pthread debug library** functions, except pthdb_session_pthreaded(), in return. Whenever the **pthread debug library** invokes a call back function, it will pass the unique application assigned user identifier back to the application. The **pthdb_session_init()** function checks the list of call back functions (“Multi-Threaded Call Back Functions” on page 186) provided by the application, and initializes the session’s data structures. Also, this function sets the session flags. An appplication must pass the **PTHDB_FLAG_SUSPEND** flag to the **pthdb_session_init**, see the pthdb_session_setflags() function for a full list of flags.

Call Back Functions

The **pthread debug library** uses the call back functions to to obtain data, to write data, and to give storage management to the application. See Call Back Functions (“Multi-Threaded Call Back Functions” on page 186) for more information.

Required call back functions for an application:

- read_data - needed to retrieve **pthread library** object information
- alloc - needed to alloc memory in the **pthread debug library**
- realloc - needed to re-alloc memory in the **pthread debug library**
- dealloc - needed to free allocated memory in the **pthread debug library**

Optional call back functions for an application:

- read_regs - only necessary for **pthdb_pthread_context** and **pthdb_pthread_setcontext**.
- write_data - only necessary for **pthdb_pthread_setcontext**.

- write_regs - only necessary for **pthdb_pthread_setcontext**.

Update Function

Each time the application is stopped, after the session has been initialized, it is necessary to call the pthdb_session_update() function. This function sets or reset the lists of pthreads, pthread attributes, mutexes, mutex attributes, condition variables, condition variable attributes, read/write locks, read/write lock attributes, pthread specific keys and active keys. It uses call back functions to manage memory for the lists.

Context Functions

The pthdb_pthread_context() function is used to get the context information and the pthdb_pthread_setcontext() function is used to set the context. The **pthdb_pthread_context()** function obtains the context information of a pthread from either the kernel or the pthread data structure in the application's address space. If the pthread is not associated with a kernel thread, then the context information saved by **pthread library** is obtained. If a pthread is associated with a kernel thread, the information is obtained from the application using the call back functions, it is the applications responsibility to determine if the kernel thread is in kernel mode or user mode and provide the correct information for that mode.

When a pthread with kernel thread is in kernel mode code it is impossible to get the full user mode context because the kernel does not save it off in one place. The **getthrds()** function can be used to get part of this information. It always saves the user mode stack and the application can discover this by checking **thrdsinfo64.ti_scount**. If this is non-zero the user mode stack is available in **thrdsinfo64.ti_ustk**. From user mode stack it is possible to determine the iar and the call back frames but not the other register values. The **thrdsinfo64** structure is defined in **procinfo.h** file.

List Functions

The **pthread debug library** maintains lists for pthreads, pthread attributes, mutexes, mutex attributes, condition variables, condition variables attributes, read/write locks, read/write lock attributes, pthread specific keys and active keys, each represented by a type specific handle. The pthdb_<object>() functions return the next handle in the appropriate list, where object is one of the following: **pthread, attr, mutex, mutexattr, cond, condattr, rwlock, rwlockattr** or **key**. If the list is empty or the end of the list is reached, **PTHDB_INVALID_<OBJECT>** is reported, where OBJECT is one of the following: **PTHREAD, ATTR, MUTEX, MUTEXATTR, COND, CONDATTR, RWLOCK, RWLOCKATTR** or **KEY**.

Field Functions

Detailed information about an object can be obtained by using the appropriate object member function, **pthdb_<object>_<field>**(), where object is one of the following: **pthread, attr, mutex, mutexattr, cond, condattr, rwlock, rwlockattr** or **key** and where field is the name of a field of the detailed information for the object.

Customizing the Session

The pthdb_session_setflags() function allows the application to change the flags which customize the session. These flags are used to control the number of registers that are read or wrote during context operations.

The pthdb_session_flags() function gets the current flags for the session.

Session Termination

At the end of the session, the session data structures need to be deallocated and the session data needs to be deleted. This is accomplished by calling the pthdb_session_destroy() function, which uses a call back function to deallocate the memory. All of the memory allocated by the pthdb_session_init(), and pthdb_session_update() functions will be deallocated.

Example

Pseudo-code showing how an application can connect to the **pthread debug library**:

```
/* includes */

#include <pthread.h>
#include <sys/pthdebug.h>

...

int my_read_data(pthdb_user_t user, pthdb_symbol_t symbols[],int count)
{
  int rc;

  rc=memcpy(buf,(void *)addr,len);
  if (rc==NULL) {
    fprintf(stderr,"Error message\n");
    return(1);
  }
  return(0);
}
int my_alloc(pthdb_user_t user, size_t len, void **bufp)
{
  *bufp=malloc(len);
  if(!*bufp) {
    fprintf(stderr,"Error message\n");
    return(1);
  }
  return(0);
}
int my_realloc(pthdb_user_t user, void *buf, size_t len, void **bufp)
{
  *bufp=realloc(buf,len);
  if(!*bufp) {
    fprintf(stderr,"Error message\n");
    return(1);
  }
  return(0);
}
int my_dealloc(pthdb_user_t user,void *buf)
{
  free(buf);
  return(0);
}

status()
{
  pthdb_callbacks_t callbacks =
                    {  NULL,
                       my_read_data,
                       NULL,
                       NULL,
                       NULL,
                       my_alloc,
                       my_realloc,
                       my_dealloc,
                       NULL
                    };

  ...

  rc=pthread_suspend_others_np();
  if (rc!=0)
    deal with error

  if (not initialized)
```

```
    rc=pthdb_session_init(user,exec_mode,PTHDB_SUSPEND|PTHDB_REGS,callbacks,
                          &session);
    if (rc!=PTHDB_SUCCESS)
       deal with error

  rc=pthdb_session_update(session);
  if (rc!=PTHDB_SUCCESS)
        deal with error

   retrieve pthread object information using the object list functions and
   the object field functions

  ...

  rc=pthread_continue_others_np();
  if (rc!=0)
    deal with error
}

...

main()
{
  ...
}
```

Related Information

- Session Functions
  - pthdb_session_concurrency
  - pthdb_session_destroy
  - pthdb_session_flags
  - pthdb_session_setflags
  - pthdb_session_init
  - pthdb_session_update
- Call Back Functions (“Multi-Threaded Call Back Functions” on page 186)
  - read_data
  - write_data
  - read_regs
  - write_regs
  - alloc
  - realloc
  - dealloc
- List Functions
  - pthdb_attr
  - pthdb_cond
  - pthdb_condattr
  - pthdb_key
  - pthdb_mutex
  - pthdb_mutexattr
  - pthdb_pthread
  - pthdb_pthread_key
  - pthdb_rwlock
  - pthdb_rwlockattr

- Pthread Functions
  - pthdb_pthread_addr
  - pthdb_pthread_arg
  - pthdb_pthread_cancelpend
  - pthdb_pthread_cancelstate
  - pthdb_pthread_canceltype
  - pthdb_pthread_detachstate
  - pthdb_pthread_exit
  - pthdb_pthread_func
  - pthdb_pthread_ptid
  - pthdb_pthread_schedparam
  - pthdb_pthread_schedpolicy
  - pthdb_pthread_schedpriority
  - pthdb_pthread_scope
  - pthdb_pthread_state
  - pthdb_pthread_suspendstate
  - pthdb_ptid_pthread
- Pthread Context Functions
  - pthdb_pthread_context
  - pthdb_pthread_setcontext
- Pthread Signal Functions
  - pthdb_pthread_sigmask
  - pthdb_pthread_sigpend
  - pthdb_pthread_sigwait
- Pthread Specific Data Functions
  - pthdb_pthread_specific
- Pthread Mapping to Kernel Thread Functions
  - pthdb_pthread_tid
  - pthdb_tid_pthread_tid
- Attribute Functions
  - pthdb_attr_addr
  - pthdb_attr_detachstate
  - pthdb_attr_guardsize
  - pthdb_attr_inheritsched
  - pthdb_attr_schedparam
  - pthdb_attr_schedpolicy
  - pthdb_attr_schedpriority
  - pthdb_attr_scope
  - pthdb_attr_stackaddr
  - pthdb_attr_stacksize
  - pthdb_attr_suspendstate
- Mutex Functions
  - pthdb_mutex_addr
  - pthdb_mutex_lock_count
  - pthdb_mutex_owner

  - pthdb_mutex_pshared
  - pthdb_mutex_prioceiling
  - pthdb_mutex_protocol
  - pthdb_mutex_state
  - pthdb_mutex_type
- Mutex Attribute Functions
  - pthdb_mutexattr_addr
  - pthdb_mutexattr_prioceiling
  - pthdb_mutexattr_protocol
  - pthdb_mutexattr_pshared
  - pthdb_mutexattr_type
- Condition Variable Functions
  - pthdb_cond_addr
  - pthdb_cond_mutex
  - pthdb_cond_pshared
- Condition Variable Attribute Functions
  - pthdb_condattr_addr
  - pthdb_condattr_pshared
- Read/Write Lock Functions
  - pthdb_rwlock_addr
  - pthdb_rwlock_lock_count
  - pthdb_rwlock_owner
  - pthdb_rwlock_pshared
  - pthdb_rwlock_state
- Read/Write Lock Attribute Functions
  - pthdb_rwlockattr_addr
  - pthdb_rwlockattr_pshared
- Waiter Functions
  - pthdb_mutex_waiter
  - pthdb_cond_waiter
  - pthdb_rwlock_read_waiter
  - pthdb_rwlock_write_waiter

The pthread.h file

"Developing Multi-Threaded Programs" on page 173

Developing Multi-Threaded Program Debuggers

The **pthread debug library** (**libpthdebug.a**) provides a set of functions which will allow debugger developers to provide debug capabilities for applications using the **pthread library**.

This library is used to debug both 32-bit and 64-bit pthreaded applications. This library is used to debug targeted debug processes only, it can also be used introspectively (for example: linked to an application that uses pthreads) to examine pthread information of its own application. See "Developing Multi-Threaded Program which examines and modifies pthread library objects" on page 176. This library can be used by a

multi-threaded debugger to debug a multi-threaded application. Multi-threaded debuggers are supported via libpthreads.a. This library is thread safe. The pthread debug library contains a 32-bit shared object and a 64-bit shared object.

Debuggers using the ptrace facility must link to the 32-bit version of the library, since the ptrace facility is not suppoted in 64-bit mode. Debuggers using the /proc facility can link to either the 32-bit version or the 64-bit version of this library.

The **pthread debug library** provides debuggers access to **pthread library** information. This includes information on pthreads, pthread attributes, mutexes, mutex attributes, condition variables, condition variable attributes, read/write locks, read/write lock attributes, and information about the state of the **pthread library**. This library also provides help with controlling the execution of pthreads.

Note: All data (addresses, registers) returned by this library will be in 64-bit format both for 64-bit and 32-bit application. It is the debuggers responsibility to convert these values into 32-bit format for 32-bit applications. When debugging a 32-bit application the top half of addresses and registers will be ignored.

Note: The pthread debug library does not report mutexes, mutexattrs, conds, condattrs, rwlocks, rwlockattrs that have the pshared value of PTHREAD_PROCESS_SHARED.

Initialization

The debugger must initialize a **pthread debug library** session for each debug process. This cannot be done until the **pthread library** has been initialized in the debug process. The **pthdb_session_pthreaded()** function has been provided to tell the debugger when the **pthread library** has been initialized in the debug process. Each time, the **pthdb_session_pthreaded()** function is called it checks to see if the **pthread library** has been initialized. If initialized, it returns `PTHDB_SUCCESS`. Otherwise it returns `PTHDB_NOT_PTHREADED`. In both cases, it returns a function name which can be used to set a breakpoint for immediate notification that the **pthread library** has been initialized. Therefore, the **pthdb_session_pthreaded()** function provides two methods to determine when the **pthread library** has been initialized:

- The first method requires the debugger to call the function each time the debug process stops, to see if the debuggee is pthreaded.
- The second method requires the debugger to call the function once and if the debuggee is not pthreaded to set a breakpoint to notify the debugger when the debug process is pthreaded.

Once the debug process is pthreaded, the debugger must call the **pthdb_session_init()** function, to initialize a session for the debug process. The **pthread debug library** supports one session for a single debug process. The debugger must assign a unique user identifier and pass it to **pthdb_session_init()** which in turn will assign a unique session identifier which must be passed as the first parameter to all other **pthread debug library** functions, except **pthdb_session_pthreaded()**, in return. Whenever the **pthread debug library** invokes a “Multi-Threaded Call Back Functions” on page 186, it will pass the unique debugger assigned user identifier back to the debugger. The **pthdb_session_init()** function checks the list of call back functions provided by the debugger, and initializes the session’s data structures. Also, this function sets the session flags, see the **pthdb_session_setflags** function.

Call Back Functions

The **pthread debug library** uses call back functions to obtain addresses, to obtain data, to write data, to give storage management to the debugger, and to aid in debugging the **pthread debug library**. See “Multi-Threaded Call Back Functions” on page 186 for more information.

Update Function

Each time the debugger is stopped, after the session has been initialized, it is necessary to call the **pthdb_session_update()** function. This function sets or reset the lists of pthreads, pthread attributes,

mutexes, mutex attributes, condition variables, condition variable attributes, read/write locks, read/write lock attributes, pthread specific keys and active keys. It uses call back functions to manage memory for the lists.

Hold and Unhold Functions

Debuggers need to support hold and unhold of threads for two reasons:

- In order to allow a user to single step a single thread, it must be possible to hold one or more of the other threads.
- For users to continue a subset of available threads, it must be possible to hold threads not in the set.

The **pthdb_pthread_hold()** function sets the *hold state* of a pthread to `hold`.

The **pthdb_pthread_unhold()** function sets the *hold state* of a pthread to `unhold`.

Note: The **pthdb_pthread_hold()** and **pthdb_pthread_unhold()** functions must always be used whether a pthread has a kernel thread or not.

The **pthdb_pthread_holdstate()** function returns the *hold state* of the pthread.

The **pthdb_session_committed()** function reports the function name of the function that is called after all of the hold and unhold changes are committed. A break point can be placed at this function to notify the debugger when the hold and unhold changes have been committed.

The **pthdb_session_stop_tid()** function informs the **pthread debug library**, which informs the **pthread library** the tid of the thread that stopped the debugger.

The **pthdb_session_commit_tid()** function returns the list of kernel threads, one kernel thread at a time, that must be continued to commit the hold and unhold changes. This function must be called repeatedly, until `PTHDB_INVALID_TID` is reported. If the list of kernel threads is empty, it is not necessary to continue any threads for the commit operation.

The debugger can determine when all of the hold and unhold changes have been committed in two ways:

- Before the commit operation (continuing all of the tids returned by the **pthdb_session_commit_tid()** function) is started, the debugger can call the **pthdb_session_committed()** function to get the function name and set a breakpoint. (This method can be done once for the life of the process.)
- Before the commit operation is started, the debugger calls the **pthdb_session_stop_tid()** function with the tid of the thread that stopped the debugger. When the commit operation is complete, the **pthread library** will ensure that the same stop tid is stopped as before the commit operation.

In order to hold or unhold pthreads it is necessary to follow the following procedure, before continuing a group of pthreads or single stepping a single pthread:

1. Use the **pthdb_pthread_hold()** and **pthdb_pthread_unhold()** functions to set up which pthreads will be held and which will be unheld.
2. Set-up the method that will determine when all of the hold and unhold changes have been committed.
3. Use the **pthdb_session_commit_tid()** function to determine the list of tids that must be continued to commit the hold and unhold changes.
4. Continue the tids in the previous step and the thread which stopped the debugger.

The **pthdb_session_continue_tid()** function allows the debugger to obtain the list of kernel threads that must be continued before it proceeds with single stepping a single pthread or continuing a group of pthreads. This function must be called repeatedly, until `PTHDB_INVALID_TID` is reported. If the list of kernel threads is not empty, the debugger will need to continue these kernel threads along with the others it is

explicitly interested in. The debugger is responsible for parking the stop thread and continuing the stop thread. The stop thread, is the thread that caused the debugger to be entered.

Context Functions

The **pthdb_pthread_context()** function is used to get the context information and the **pthdb_pthread_setcontext()** function is used to set the context. The **pthdb_pthread_context()** function obtains the context information of a pthread from either the kernel or the pthread data structure in the debug process's address space. If the pthread is not associated with a kernel thread, then the context information saved by **pthread library** is obtained. If a pthread is associated with a kernel thread, the information is obtained from the debugger using call backs, it is the debuggers responsibility to determine if the kernel thread is in kernel mode or user mode and provide the correct information for that mode.

When a pthread with kernel thread is in kernel mode code it is impossible to get the full user mode context because the kernel does not save it off in one place. The **getthrds()** function can be used to get part of this information. It always saves the user mode stack and the debugger can discover this by checking *thrdsinfo64.ti_scount*. If this is non-zero the user mode stack is available in **thrdsinfo64.ti_ustk**. From user mode stack it is possible to determine the **iar** and the call back frames but not the other register values. The thrdsinfo64 structure is defined in **procinfo.h** file.

List Functions

The **pthread debug library** maintains lists for pthreads, pthread attributes, mutexes, mutex attributes, condition variables, condition variables attributes, read/write locks, read/write lock attributes, pthread specific keys and active keys, each represented by a type specific handle. The **pthdb_<*object*>()** The pthdb_<object>() functions return the next handle in the appropriate list, where <object> is one of the following: pthread, attr, mutex, mutexattr, cond, condattr, rwlock, rwlockattr or key. If the list is empty or the end of the list is reached, **PTHDB_INVALID_*object*** is reported, where ***object*** is one of the following: `PTHREAD`, `ATTR`, `MUTEX`, `MUTEXATTR`, `COND`, `CONDATTR`, `RWLOCK`, `RWLOCKATTR` or `KEY`.

Field Functions

Detailed information about an object can be obtained by using the appropriate object member function, **pthdb_*object*_*field*()**, where ***object*** is one of the following: `pthread`, `attr`, `mutex`, `mutexattr`, `cond`, `condattr`, `rwlock`, `rwlockattr` or `key` and where ***field*** is the name of a field of the detailed information for the object.

Customizing the Session

The **pthdb_session_setflags()** function allows the debugger to change the flags which customize the session. These flags are used to control the number of registers that are read or wrote during context operations, and to control the printing of debug information.

The **pthdb_session_flags()** function gets the current flags for the session.

Session Termination

At the end of the debug session, the session data structures need to be deallocated and the session data needs to be deleted. This is accomplished by calling the **pthdb_session_destroy()** function, which uses a call back functions to deallocate the memory. All of the memory allocated by the **pthdb_session_init()**, and **pthdb_session_update()** functions will be deallocated.

Example

Pseudo-code showing how the debugger should make use of the hold/unhold code:

```
/* includes */

#include <sys/pthdebug.h>
```

```
main()
{
    tid_t stop_tid; /* thread which stopped the process */
    pthdb_user_t user = <unique debugger value>;
    pthdb_session_t session; /* <unique library value> */
    pthdb_callbacks_t callbacks = <callback functions>;
    char *pthreaded_symbol=NULL;
    char *committed_symbol;
    int pthreaded = 0;
    int pthdb_init = 0;
    char *committed_symbol;

    /* fork/exec or attach to debuggee */

    /* debuggee uses ptrace()/ptracex() with PT_TRACE_ME */

    while (/* waiting on an event */)
    {
      /* debugger waits on debuggee */

      if (pthreaded_symbol==NULL) {
        rc = pthdb_session_pthreaded(user, &callbacks, pthreaded_symbol);
        if (rc == PTHDB_NOT_PTHREADED)
        {
            /* set breakpoint at pthreaded_symbol */
        }
        else
          pthreaded=1;
      }
      if (pthreaded == 1 && pthdb_init == 0) {
          rc = pthdb_session_init(user, &session, PEM_32BIT, flags, &callbacks);
          if (rc)
              /* handle error and exit */
          pthdb_init=1;
      }

      rc = pthdb_session_update(session)
      if ( rc != PTHDB_SUCCESS)
/* handle error and exit */

      while (/* accepting debugger commands */)
      {
          switch (/* debugger command */)
          {
              ...
              case DB_HOLD:
                  /* regardless of pthread with or without kernel thread */
                  rc = pthdb_pthread_hold(session, pthread);
                  if (rc)
                      /* handle error and exit */
              case DB_UNHOLD:
                  /* regardless of pthread with or without kernel thread */
                  rc = pthdb_pthread_unhold(session, pthread);
                  if (rc)
                      /* handle error and exit */
              case DB_CONTINUE:
                  /* unless we have never held threads for the life */
                  /* of the process */
                  if (pthreaded)
                  {
                      /* debugger must handle list of any size */
                      struct pthread commit_tids;
                      int commit_count = 0;
                      /* debugger must handle list of any size */
                      struct pthread continue_tids;
                      int continue_count = 0;
```

```
      rc = pthdb_session_committed(session, committed_symbol);
      if (rc != PTHDB_SUCCESS)
  /* handle error */
              /* set break point  at committed_symbol */

                      /* gather any tids necessary to commit hold/unhold */
                      /* operations */
                      do
                      {
                          rc = pthdb_session_commit_tid(session,
                                              &commit_tids.th[commit_count++]);
                          if (rc != PTHDB_SUCCESS)
                              /* handle error and exit */
                      } while (commit_tids.th[commit_count - 1] != PTHDB_INVALID_TID);

                      /* set up thread which stopped the process to be */
                      /* parked using the stop_park function*/

      if (commit_count > 0) {
                        rc = ptrace(PTT_CONTINUE, stop_tid, stop_park, 0,
                                                            &commit_tids);
                        if (rc)
                            /* handle error and exit */

                        /* wait on process to stop */
      }

                      /* gather any tids necessary to continue */
                      /* interesting threads */
                      do
                      {
                          rc = pthdb_session_continue_tid(session,
                                          &continue_tids.th[continue_count++]);
                           if (rc != PTHDB_SUCCESS)
                              /* handle error and exit */
                      } while (continue_tids.th[continue_count - 1] != PTHDB_INVALID_TID);

                      /* add interesting threads to continue_tids */

                      /* set up thread which stopped the process to be parked */
                      /* unless it is an interesting thread */

                      rc = ptrace(PTT_CONTINUE, stop_tid, stop_park, 0,
                                                              &continue_tids);
                      if (rc)
                          /* handle error and exit */
                  }
              case DB_EXIT:
rc = pthdb_session_destroy(session);
/* other clean up code */
exit(0);
              ...
          }
      }

    }
    exit(0);
}
```

Multi-Threaded Call Back Functions

- symbol_addrs
- read_data
- write_data
- read_regs

- write_regs
- alloc
- realloc
- dealloc
- print

Purpose

Provide access to facilities needed by the pthread debug library and supplied by the debugger or application.

Library

These functions are provided by the debugger which links in the *pthread debug library*.

Syntax

```
#include <sys/pthdebug.h>

int symbol_addrs(pthdb_user_t     user,
                 pthdb_symbol_t   symbols[],
                 int              count)

int read_data(pthdb_user_t     user,
              void            * buf,
              pthdb_addr_t     addr,
              int              size)

int write_data(pthdb_user_t     user,
               void            * buf,
               pthdb_addr_t     addr,
               int              size)

int read_regs(pthdb_user_t     user,
              tid_t               tid,
              unsigned long long  flags,
              struct context64   * context)

int write_regs(pthdb_user_t     user,
               tid_t               tid,
               unsigned long long  flags,
               struct context64   * context)

int alloc(pthdb_user_t     user,
          size_t           len,
          void            ** bufp)

int realloc(pthdb_user_t     user,
            void            * buf,
            size_t           len,
            void            ** bufp)

int dealloc(pthdb_user_t     user,
            void            * buf)
```

```
int print(pthdb_user_t     user,
          char            * str)
```

Description

int symbol_addrs()

Resolves the address of symbols in the debuggee. The pthread debug library will call this function to get the address of known debug symbols. If symbol has a name of NULL or ″″, then just set the address to 0LL, instead of doing a lookup or returning an error. If successful, 0 is returned, else non-zero is returned. In introspective mode, when the PTHDB_FLAG_SUSPEND flag is set, the application can use the pthread debug library provided symbol_addrs call back function, by passing NULL or it can use one of it's own.

int read_data()

Reads the requested number of bytes of data at requested location from an active process or from a core file and returns the data through a buffer. If successful then return 0 else return non-zero. This call back function is always required.

int write_data()

Writes the requested number of bytes of data at requested location. The pthdebug library may use this to write data to the active process. If successful return 0, else non-zero is returned. This call back function is required when the PTHDB_FLAG_HOLD flag is set and when using the pthdb_pthread_setcontext() function.

int read_regs()

Read registers call back function should be able to read the context information of a debuggee kernel thread from an active process or from a core file. The information should be formatted in context64 form for both 32-bit and 64-bit process. If successful return 0, else non-zero is returned. This function is only required when using the pthdb_pthread_context() and pthdb_pthread_setcontext() functions.

int write_regs()

Write register function should be able to write requested context information to specified debuggee's kernel thread id. If successful return 0, else non-zero is returned. This function is only required when using the pthdb_pthread_setcontext() functions.

int alloc()

Takes len and allocates len bytes of memory and returns the address. If successful return 0, else non-zero is returned. This call back function is always required.

int realloc()

Takes len and the buf and re-allocates the memory and returns an address to the realloc memory. If successful return 0, else non-zero is returned. This call back function is always required.

int dealloc()

Takes a buffer and frees it. If successful return 0, else non-zero is returned. This call back function is always required.

int print()

Prints the character string on the debugger's stdout. If successful return 0, else non-zero is returned. This call back is for **debugging the library only**, the messages printed will not be translated and will not be explained in our user level documentation. If not debugging the *pthread debug library* pass a NULL value for this call back.

Note: If **write_data()** and **write_regs()** are NULL then the **pthread debug library** will not try to write data or write regs. If *pthdb_pthread_set_context()* is called when **write_data()** and **write_regs()** are NULL, then it will return *PTHDB_NOTSUP*.

Parameters

user	User handle.
symbols	Array of symbols.
count	Number of symbols.
buf	Buffer.
addr	Address to be read from or wrote to.
size	Size of buffer.
flags	Session flags, must accept *PTHDB_FLAG_GPRS*, *PTHDB_FLAG_SPRS*, *PTHDB_FLAG_FPRS* and *PTHDB_FLAG_REGS*.
tid	Thread id.
flags	Flags which control which registers are read or wrote.
context	Context structure.
len	Length of buffer to be allocated or re-allocated.
bufp	Pointer to buffer.
str	String to be printed.

Return Values

If successful these function returns 0 else returns a non-zero value.

Related Information

The **pthdebug.h** file.

Benefits of Threads

The following explains the benefits of writing multi-threaded programs. Major improvements of threads programming are:

- "Parallel Programming Concepts" are easier to implement.
- Multi-threaded programs provide better performance. See "Performance Consideration" on page 191.

Threads do have some "Limitations" on page 191 and cannot be used for some special purposes which still require multi-processed programs.

Parallel Programming Concepts

There are two main advantages for using parallel programming instead of serial programming techniques:

- Parallel programming can improve the performance of a program.
- Some common software models are well suited to parallel programming techniques.

Traditionally, multiple single-threaded processes have been used to achieve parallelism, but some programs can benefit from a finer level of parallelism. Multi-threaded processes offer parallelism within a process and share many of the concepts involved in programming multiple single-threaded processes.

Modularity

Programs are often modeled as a number of distinct parts interacting with each other to produce a desired result or service. A program can be implemented as a single, complex entity that performs multiple functions among the different parts of the program. A more simple solution consists of implementing several entities, each entity performing a part of the program and sharing resources with other entities.

By using multiple entities, a program can be separated according to its distinct activities, each having an associated entity. These entities do not have to know anything about the other parts of the program except when they exchange information. In these cases, they must synchronize with each other to ensure data integrity.

Threads are well-suited entities for modular programming. Threads provide simple data sharing (all threads within a process share the same address space) and powerful synchronization facilities (such as mutexes and condition variables).

Software Models

The following common software models can easily be implemented with threads.

- "Master/Slave Model"
- "Divide-and-Conquer Models"
- "Producer/Consumer Models".

All these models lead to modular programs. Models may also be combined to efficiently solve complex tasks.

These models can apply to either traditional multi-process solutions, or to single process multi-thread solutions, on multi-threaded systems such as AIX. In the following descriptions, the word *entity* refers to either a single-threaded *process* or to a single *thread* in a multi-threaded process.

Master/Slave Model

In the master/slave (sometimes called boss/worker) model, a master entity receives one or more requests, then creates slave entities to execute them. Typically, the master controls how many slaves there are and what each slave does. A slave runs independently of other slaves.

An example of this model is a print job spooler controlling a set of printers. The spooler's role is to ensure that the print requests received are handled in a timely fashion. When the spooler receives a request, the master entity chooses a printer and causes a slave to print the job on the printer. Each slave prints one job at a time on a printer, handling flow control and other printing details. The spooler may support job cancellation or other features which require the master to cancel slave entities or reassign jobs.

Divide-and-Conquer Models

In the divide-and-conquer (sometimes called simultaneous computation or work crew) model, one or more entities perform the same tasks in parallel. There is no master entity; all entities run in parallel independently.

An example of a divide-and-conquer model is a parallelized **grep** command implementation, which could be done as follows. The **grep** command first establishes a pool of files to be scanned. It then creates a number of entities. Each entity takes a different file from the pool and searches for the pattern, sending the results to a common output device. When an entity completes its file search, it obtains another file from the pool or stops if the pool is empty.

Producer/Consumer Models

The producer/consumer (sometimes called pipelining) model is typified by a production line. An item proceeds from raw components to a final item in a series of stages. Usually a single worker at each stage modifies the item and passes it on to the next stage. In software terms, an AIX command pipe, such as the **cpio** command, is a good example of a this model.

For example, a Reader entity reads raw data from standard input and passes it to the processor entity, which processes the data and passes it to the writer entity, which writes it to standard output. Parallel programming allows the activities to be performed concurrently: the writer entity may output some processed data while the reader entity gets more raw data.

Performance Consideration

Multi-threaded programs can improve performance in many ways compared to traditional parallel programs using multiple processes. Furthermore, higher performance can be obtained on multiprocessor systems using threads.

Managing Threads

Managing threads, that is creating threads and controlling their execution, requires fewer system resources than managing processes. Creating a thread, for example, only requires the allocation of the thread's private data area, usually 64KB, and two system calls. Creating a process is far more expensive, because the entire parent process addressing space is duplicated.

The threads library API is also easier to use than the one for managing processes. Programmers should think about the six ways of calling the **exec** subroutine. Thread creation requires just one syntax: the **pthread_create** subroutine.

Inter-Thread Communications

Inter-thread communication is far more efficient and easier to use than inter-process communication. Because all threads within a process share the same address space, they need not use shared memory. Shared data should just be protected from concurrent access using mutexes or other synchronization tools.

Synchronization facilities provided by the threads library allow easy implementation of flexible and powerful synchronization tools. These tools can easily replace traditional inter-process communication facilities, such as message queues. Note that pipes can be used as an inter-thread communication path.

Multiprocessor Systems

On a multiprocessor system, multiple threads can concurrently run on multiple CPUs. Therefore, multi-threaded programs can run much faster than on a uniprocessor system. They will also be faster than a program using multiple processes, because threads require fewer resources and generate less overhead. For example, switching threads in the same process can be faster, especially in the M:N library model where context switches can often be avoided. Finally, a major advantage of using threads is that a single multi-threaded program will work on a uniprocessor system, but can naturally take advantage of a multiprocessor system, without recompiling.

Limitations

Multi-threaded programming is useful for implementing parallelized algorithms using several independent entities. However, there are some cases where multiple processes should be used instead of multiple threads.

Many operating system identifiers, resources, states, or limitations are defined at the process level and, thus, are shared by all threads in a process. For example, user and group IDs and their associated permissions are handled at process level. Programs that need to assign different user IDs to their programming entities need to use multiple processes, instead of a single multi-threaded process. Other examples include file system attributes such as the current working directory, and the state and maximum number of open files. Multi-threaded programs may not be appropriate if these attributes are better handled independently. For example, a multi-processed program can let each process open a large number of files without interference from other processes.

Chapter 10. Programming on Multiprocessor Systems

On a uniprocessor system, threads execute one after another in a time-sliced manner. This contrasts with a multiprocessor system, where several threads execute at the same time, one on each available processor. Overall performance is improved by running different process threads on different processors. However, an individual program cannot take advantage of multiprocessing, unless it has multiple threads.

For most users, multiprocessing is invisible, being completely handled by the operating system and the programs it runs. If desired, users may bind their processes (force them to run on a certain processor); however, this is not required, nor recommended for ordinary use. Even for most programmers, taking advantage of multiprocessing simply amounts to using multiple threads. On the other hand, kernel programmers have to deal with several issues when porting or creating code for multiprocessor systems. The following information discusses these topics.

Identifying Processors

Symmetric multiprocessor machines have one or more CPU boards, each of which can accommodate two processors. For example, a four processor machine has two CPU boards, each having two processors. Commands, subroutines, or messages that refer to processors need to use an identification scheme. Processors are identified by physical and logical numbers, and by Object Data Manager (ODM) processor names and location codes.

ODM Processor Names

ODM is a system used to identify various parts throughout a machine, including bus adapters, peripherals such as printers or terminals, disks, memory boards, and processor boards. See “Chapter 17. Object Data Manager (ODM)” on page 507 for more information about ODM.

ODM assigns numbers to processor boards and processors in order, starting from 0 (zero), and creates names based on these numbers by adding a prefix `cpucard` or `proc`. Thus, the first processor board is called `cpucard0`, and the second processor on it is called `proc1`.

ODM location codes for processors consist of four 2-digit fields, in the form *AA-BB-CC-DD*, as explained below:

AA Always `00`. It indicates the main unit.
BB Indicates the processor board number. It can be `0P`, `0Q`, `0R`, or `0S`, indicating respectively the first, second, third or fourth processor card.
CC Always `00`.
DD Indicates the processor position on the processor board. It can be `00` or `01`.

These codes are illustrated in “Example Processor Configurations” on page 194.

Logical Processor Numbers

Processors can also be identified using logical numbers, which start with 0 (zero). Only enabled processors have a logical number.

The logical processor number `0` (zero) identifies the first physical processor in the enabled state; the logical processor number `1` (one) identifies the second enabled physical processor, and so on.

Generally, all operating system commands and library subroutines use logical numbers to identify processors. The **cpu_state** command (see "The cpu_state Command") is an exception and uses ODM processor names instead of logical processor numbers.

ODM Processor States

If a processor functions correctly, it can be enabled or disabled using a software command. A processor is marked **faulty** if it has a detected hardware problem. ODM classifies processors using three states. A processor can only be in one of the following states:

enabled	Processor works and can be used by AIX.
disabled	Processor works, but cannot be used by AIX.
faulty	Processor does not work (a hardware fault was detected).

Controlling Processor Use

On a multiprocessor system, the use of processors can be controlled in two ways:

- A system administrator can control the availablity of the processors for the whole system.
- A user can force a process or kernel threads to run on a specific processor.

The cpu_state Command

A system administrator (or any user with root authority) can use the **cpu_state** command to list system processors or to control available processors. This command can be used to list the following information for each configured processor in the system:

Name	"ODM Processor Names" on page 193, shown in the form `procx`, where *x* is the physical processor number
Cpu	"Logical Processor Numbers" on page 193
Status	"ODM Processor States" for the next boot
Location	"ODM Processor Names" on page 193, shown in the form *AA-BB-CC-DD*

> **Note:** The **cpu_state** command does not display the current processor state, but instead displays the state to be used for the next system start up (enabled or disabled). If the processor does not respond, it is either **faulty** (an ODM state) or a communication error occured. In this case, the **cpu_state** command displays `No Reply`.

Example Processor Configurations

The examples that follow show various processor configurations and how they affect the output of the **cpu_state** command. These examples illustrate the relationships among physical processor numbers, logical processor numbers, the current processor state, and the processor state used at the next boot.

Simple Processor Configurations

The simplest case to consider is when all available processors on a system are functionning and enabled. Consider a simple two processor system with both processors enabled. The various ODM and number conventions are shown in the following table.

Processor Naming Conventions				
ODM Card Name	**ODM Processor Name**	**Logical Number**	**ODM Current Processor State**	**cpu_state Status Field**
cpucard0	proc0	0	Enabled	Enabled
cpucard0	proc1	1	Enabled	Enabled

For the above configuration, the **cpu_state -l** command produces a listing similar to the following:

```
Name    Cpu      Status     Location
proc0    0       Enabled    00-0P-00-00
proc1    1       Enabled    00-0P-00-01
```

The following example shows the system upgraded by adding an additional CPU card with two processors. By default, processors are enabled, so the new configuration is shown in the following table.

Processor Naming Conventions				
ODM Card Name	**ODM Processor Name**	**Logical Number**	**ODM Current Processor State**	**cpu_state Status Field**
cpucard0	proc0	0	Enabled	Enabled
cpucard0	proc1	1	Enabled	Enabled
cpucard1	proc2	2	Enabled	Enabled
cpucard1	proc3	3	Enabled	Enabled

For this configuration, the **cpu_state -l** command produces a listing similar to the following:

```
Name    Cpu      Status     Location
proc0    0       Enabled    00-0P-00-00
proc1    1       Enabled    00-0P-00-01
proc2    2       Enabled    00-0Q-00-00
proc3    3       Enabled    00-0Q-00-01
```

Complex Processor Configurations

In some conditions, a processor is not enabled and does not have a logical processor number. A processor can fail a boot power-on test and be marked **faulty** by ODM. A processor can also be disabled for maintenance or test reasons. Also, when a processor is enabled or disabled using the **cpu_state** command, its current state remains unchanged until the next boot, but its state at the next boot (displayed in the **Status** field of the **cpu_state** command) is changed immediately.

Disabled Processor Configurations: Using the four processor configurations in the previous section, the physical processor 1 can be disabled with the command:

```
cpu_state -d proc1
```

The processor configuration is shown in the following table.

Processor Naming Conventions				
ODM Card Name	**ODM Processor Name**	**Logical Number**	**ODM Current Processor State**	**cpu_state Status Field**
cpucard0	proc0	0	Enabled	Enabled
cpucard0	proc1	1	Enabled	Disabled
cpucard1	proc2	2	Enabled	Enabled
cpucard1	proc3	3	Enabled	Enabled

For this configuration, the **cpu_state -l** command produces a listing similar to the following:

```
Name    Cpu      Status     Location
proc0    0       Enabled    00-0P-00-00
proc1    1       Disabled   00-0P-00-01
proc2    2       Enabled    00-0Q-00-00
proc3    3       Enabled    00-0Q-00-01
```

When the system is rebooted, the processor configuration is as shown in the following table.

Processor Naming Conventions				
ODM Card Name	**ODM Processor Name**	**Logical Number**	**ODM Current Processor State**	**cpu_state Status Field**
cpucard0	proc0	0	Enabled	Enabled
cpucard0	proc1	1	Disabled	Disabled
cpucard1	proc2	2	Enabled	Enabled
cpucard1	proc3	3	Enabled	Enabled

The output of the **cpu_state -l** command is similar to the following:

```
Name    Cpu      Status     Location
proc0    0       Enabled    00-0P-00-00
proc1    -       Disabled   00-0P-00-01
proc2    1       Enabled    00-0Q-00-00
proc3    2       Enabled    00-0Q-00-01
```

Faulty Processor Configurations: The following example uses the last processor configuration discussed in the previous sectoin. The system is rebooted with processors `proc0` and `proc3` failing their power-on tests. The processor configuration is as shown in the following table.

Processor Naming Conventions				
ODM Card Name	**ODM Processor Name**	**Logical Number**	**ODM Current Processor State**	**cpu_state Status Field**
cpucard0	proc0	-	Faulty	No Reply
cpucard0	proc1	-	Disabled	Disabled
cpucard1	proc2	0	Enabled	Enabled
cpucard1	proc3	-	Faulty	No Reply

The output of the **cpu_state -l** command is similar to the following:

```
Name    Cpu      Status     Location
proc0    -       No Reply   00-0P-00-00
proc1    -       Disabled   00-0P-00-01
proc2    0       Enabled    00-0Q-00-00
proc3    -       No Reply   00-0Q-00-01
```

Binding Processes and Kernel Threads

Users may also force their processes to run on a given processor; this action is called *binding*. A system administrator may bind any process. From the command line, binding is controlled with the **bindprocessor** command.

It is important to understand that a process itself is not bound, but rather its kernel threads are bound. Once kernel threads are bound, they are always scheduled to run on the chosen processor, unless they are later unbound. When a new kernel thread is created, it has the same bind properties as its creator.

This applies to the initial thread in the new process created by the **fork** subroutine; the new thread inherits the bind properties of the thread that called the **fork** subroutine. When the **exec** subroutine is called, bind properties are left unchanged. Once a process is bound to a processor, if no other binding or unbinding action is performed, all child processes will be bound to the same processor.

It is only possible to bind processes to enabled processors using logical processor numbers. Available logical processor numbers can be listed using the **bindprocessor -q** command. For a system with four enabled processors, this command produces output similar to:

```
The available processors are: 0 1 2 3
```

Binding may also be controlled within a program using the **bindprocessor** subroutine, which allows the programmer to bind a single kernel thread or all kernel threads in a process. The programmer can also unbind either a single kernel thread or all kernel threads in a process.

Dynamic Processor Deallocation

Starting with machine type 7044 model 270, the hardware of all systems with more than two processors will be able to detect correctable errors, which are gathered by the firmware. These errors are not fatal and, as long as they remain rare occurrences, can be safely ignored. However, when a pattern of failures seems to be developing on a specific processor, this pattern may indicate that this component is likely to exhibit a fatal failure in the near future. This prediction is made by the firmware based-on-failure rates and threshold analysis.

AIX, on these systems, implements continuous hardware surveillance and regularly polls the firmware for hardware errors. When the number of processor errors hits a threshold and the firmware recognizes that there is a distinct probability that this system component will fail, the firmware returns an error report to AIX. In all cases, AIX logs the error in the system error log. In addition, on multiprocessor systems, depending on the type of failure, AIX attempts to stop using the untrustworthy processor and deallocate it. This feature is called *Dynamic Processor Deallocation*.

At this point, the processor is also flagged by the firmware for persistent deallocation for subsequent reboots, until maintenance personnel replaces the processor.

Potential Impact to Applications

This processor decallocation is transparent for the vast majority of applications, including drivers and kernel extensions. However, you can use AIX published interfaces to determine whether an application or kernel extension is running on a multiprocessor machine, find out how many processors there are, and bind threads to specific processors.

The interface for binding processes or threads to processors uses logical CPU numbers. The logical CPU numbers are in the range [0..*N*-1] where *N* is the total number of CPUs. To avoid breaking applications or kernel extensions that assume no ″holes″ in the CPU numbering, AIX always makes it appear for applications as if it is the ″last″ (highest numbered) logical CPU to be deallocated. For instance, on an 8-way SMP, the logical CPU numbers are [0..7]. If one processor is deallocated, the total number of available CPUs becomes 7, and they are numbered [0..6]. Externally, it looks like CPU 7 has disappeared, regardless of which physical processor failed. In the rest of this description, the term CPU is used for the logical entity and the term processor for the physical entity.

Applications or kernel extensions using processes/threads binding could potentially be broken if AIX silently terminated their bound threads or forcefully moved them to another CPU when one of the processors needs to be deallocated. Dynamic Processor Deallocation provides programming interfaces so that those applications and kernel extensions can be notified that a processor deallocation is about to happen. When these applications and kernel extensions get this notification, they are responsiblefor moving their bound threads and associated resources (such as timer request blocks) away form the last logical CPU and adapt themselves to the new CPU configuration.

If, after notification of applications and kernel extensions, some of the threads are still bound to the last logical CPU, the deallocation is aborted. In this case AIX logs the fact that the deallocation has been aborted in the error log and continues using the ailing processor. When the processor ultimately fails, it

creates a total system failure. Thus, it is important for applications or kernel extensions binding threads to CPUs to get the notification of an impending processor deallocation, and act on this notice.

Even in the rare cases where the deallocation cannot go through, Dynamic Processor Deallocation still gives advanced warning to system administrators. By recording the error in the error log, it gives them a chance to schedule a maintenance operation on the system to replace the ailing component before a global system failure occurs.

Processor Deallocation: Flow of Events

The typical flow of events for processor deallocation is as follows:

1. The firmware detects that a recoverable error threshold has been reached by one of the processors.
2. AIX logs the firmware error report in the system error log, and, when executing on a machine supporting processor deallocation, start the deallocation process.
3. AIX notifies non-kernel processes and threads bound to the last logical CPU.
4. AIX waits for all the bound threads to move away from the last logical CPU. If threads remain bound, AIX eventually times out (after ten minutes)and aborts the deallocation
5. Otherwise, AIX invokes the previously registered High Availability Event Handlers (HAEHs). An HAEH may return an error that will abort the deallocation.
6. Otherwise, AIX goes on with the deallocation process and ultimately stops the failing processor.

In case of failure at any point of the deallocation, AIX logs the failure with the reason why the deallocation was aborted. The system administrator can look at the error log, take corrective action (when possible) and restart the deallocation. For instance, if the deallocation was aborted because at least one application did not unbind its bound threads, the system administrator could stop the application(s), restart the deallocation (which should go through this time) and restart the application.

Programming Interfaces

Existing AIX Interfaces Dealing with Individual Processors

The following is a list of existing interfaces:

- "Interfaces to Determine the Number if CPUs on a System"
- "Interfaces to Bind Threads to a Specific Processor" on page 199

Interfaces to Determine the Number if CPUs on a System

sysconf Subroutine: The **sysconf** subroutine returns a number of processors using:

`_SC_NPROCESSORS_CONF`: Number of processors configured.
`_SC_NPROCESSORS_ONLN`: Number of processors online.

For more information, see *sysconf Subroutine* in *AIX 5L Version 5.1 Technical Reference: Base Operating System and Extensions Volume 2*.

The value returned by **sysconf** for `_SC_NPROCESSORS_CONF`, will remain constant between reboots. The value returned for `_SC_NPROCESSORS_ONLN` will be the count of active CPUs and will be decremented every time a processor is deallocated.

_system_configuration.ncpus: **_system_configuration.ncpus** identifies the number of CPUs active on a machine. Uniprocessor (UP) machines are identified by a 1. Values greater than 1 indicate multiprocessor (MP) machines. For more information, see **systemcfg.h** File in *AIX 5L Version 5.1 Files Reference*.

Because of processor deallocations, the ncpus value may now change over time when processors are deallocated. Code that depends upon this value being a constant will probably fail. To prevent such code

from suddenly changing between uniprocessor (UP) and multiprocessor (MP) behavior, a processor will not deallocate a if only two processors are currently active. Thus, if ncpus starts with a value greater than 1, it will be guaranteed to remain greater than 1 until the next reboot.

For code that must know how many processors were originally available at boot time, a new `ncpus_cfg` field is added to `_system_configuration` table that does remain constant between reboots.

The CPUs are identified by a logical CPU number in the range [0..(ncpus-1)]. The processors also have a physical CPU number which depends on which CPU board they are on, in which order, and so on. The commands and subroutines dealing with CPU numbers always use logical CPU numbers. To ease the transition to varying numbers of CPUs, the logical CPU numbers are contiguous numbers in the range [0..(ncpus-1)] in AIX 4.3.3. The effect of this is that from a user point of view, when a processor deallocation takes place, it always looks like the highest-numbered (″last″) logical CPU is going away, regardless of which physical processor failed.

> **Note**: To avoid problems, use the `ncpus_cfg` variable to determine what the highest possible logical CPU number is for a particular system.

Interfaces to Bind Threads to a Specific Processor

The **bindprocessor** and the **bindprocessor()** programming interface allow you to bind a thread or a process to a specific CPU, designated by its logical CPU number. Both interfaces will allow you to bind threads or processes only to active CPUs. They are mentioned here because those programs which directly use the **bindprocessor()** programming interface or are bound externally by a **bindprocessor** command must be able to handle the processor deallocation.

The primary problem seen by programs that bind to a processor when a CPU has been deallocated is that requests to bind to a deallocated processor will fail. Code that issues **bindprocessor** requests should always check the return value from those requests.

For more information on these interfaces, see bindprocessor Command in *AIX 5L Version 5.1 Commands Reference, Volume 1* or bindprocessor Subroutine in *AIX 5L Version 5.1 Technical Reference: Base Operating System and Extensions Volume 1*.

Interfaces for Processor Deallocation Notification

The notification mechanism is different for user mode applications having threads bound to the last logical CPU and for kernel extensions.

Notification in User Mode

Each thread of a user mode application that is bound to the last logical CPU will be sent a new signal **SIGCPUFAIL**, which is ignored by default. These applications need to be modified to catch this new signal and dispose of the threads bound to the last logical CPU (either by unbinding them or by binding them to a different CPU).

Notification in Kernel Mode

The drivers and kernel extensions which need to be notified of an impending processor deallocation have to register a High-Availability Event Handler (HAEH) routine with the kernel. This routine will be called when a processor deallocation is imminent. An interface is also provided to unregister the HAEH before the kernel extension is unconfigured or unloaded.

Registering a High-Availability Event Handler: The kernel exports a new function to allow notification of the kernel extensions in case of events, which affect the availability of the system.

The system call is:

```
int register_HA_handler(ha_handler_ext_t *)
```

For more information on this system call, see **register_HA_handler** in *AIX 5L Version 5.1 Technical Reference: Kernel and Subsystems Volume 1*.

The return value is equal to 0 in case of success. A non zero value indicates a failure.

The argument is a pointer to a structure describing the kernel extension's high-availability event handler. This structure is defined in a new header file, named **<sys/high_avail.h>** as follows:

```
typedef struct _ha_handler_ext_ {
    int (*_fun)();        /* Function to be invoked */
    long long _data;      /* Private data for (*_fun)() */
    char        _name[sizeof(long long) + 1];
} ha_handler_ext_t;
```

The private data field `_data` is provided for the use of the kernel extension if it is needed. Whatever value given in this field at the time of registration will be passed as a parameter to the registered fuction when it is called due to a CPU predictive failure event.

The _name field is a null terminated string with a maximum length of 8 characters (not including the null character terminator) which is used to uniquely identify the kernel extension with the kernel. This name has to be unique among all the registered kernel extensions. This name appears in the detailed data area of the `CPU_DEALLOC_ABORTED` error log entry if the kernel extension returns an error when the HAEH routine is called by the kernel.

Kernel extensions should register their HAEH only once.

Invocation of the High-Availability Event Handler: Two parameters call the HAEH routine. The first one is whatever is in the `_data` field of the `ha_handler_ext_t` structure passed to `register_HA_handler`. The second parameter is a pointer to a `ha_event_t` structure defined in **<sys/high_avail.h>** as:

```
typedef struct {                    /* High-availability related event */
    uint _magic;                    /* Identifies the kind of the event */
#define HA_CPU_FAIL 0x40505546      /* "CPUF" */
    union {
        struct {                   /* Predictive processor failure */
            cpu_t dealloc_cpu;     /* Logical ID of failing processor */
                       ushort domain;         /* future extension */
            ushort nodeid;         /* future extension */
            ushort reserved3;      /* future extension */
            uint reserved[4];      /* future extension */
        } _cpu;
        /* ... */                  /* Additional kind of events -- */
        /* future extension */
    } _u;
} haeh_event_t;
```

The function should return one of the following codes, also defined in **<sys/high_avail.h>**.

```
#define HA_ACCEPTED 0     /* Positive acknowledgement */
#define HA_REFUSED -1     /* Negative acknowledgement */
```

If any of the registered extensions does not return HA_ACCEPTED, the deallocation is aborted. The HAEH routines are called in the process environment and do not need to be pinned.

If a kernel extension depends on the CPU configuration, its HAEH routine must react to the upcoming CPU deallocation. This is highly application dependent. To allow AIX to proceed with the deconfiguration, they just need to move away their threads bound to the last logical CPU, if any. Also, if they have been using timers started from bound threads, those timers will be moved to another CPU as part of the CPU deallocation. If they have any dependency on these timers being delivered to a specific CPU, they must take actions such as stopping them, and restart their timer requests when the threads are bound to a new CPU, for instance. Again, this is very much application dependent.

Canceling the Registration of a High-Availability Event Handler: To keep the system coherent, and prevent system crashes, the kernel extensions which register an HAEH must cancel the registration when they are unconfigured and are going to be unloaded. The interface is:

```
int unregister_HA_handler(ha_handler_ext_t *)
```

For more information on the system call, see **unregister_HA_handler** in *AIX 5L Version 5.1 Technical Reference: Kernel and Subsystems Volume 1*.

Returns 0 in case of success. Any non-zero return value indicates an error.

Test Environment

Hardware problems triggering a processor deallocation are, hopefully, very rare events. In order to test any of the modifications made in applications or kernel extensions to support this processor deallocation, a command is provided to trigger the deallocation of a CPU designated by its logical CPU number. The syntax is:

```
cpu_deallocate cpunum
```

where:

`cpunum` is a valid logical cpu number.

You must reboot the system to get the target processor back online. Hence, this command is provided for test purposes only and is *not* intended as a system administration tool.

Creating Locking Services

Some programmers may want to implement their own high-level locking services instead of using the standard locking services (mutexes) provided in the threads library. For example, a database product may already use a set of internally defined services; it can be easier to adapt these locking services to a new system than to adapt all the internal modules that use these services.

For this reason, AIX provides atomic locking service primitives which can be used to build higher level locking services. To create services that are multiprocessor-safe (like the standard mutex services), programmers must use the atomic locking services described in this section and not atomic operations services, such as **compare_and_swap**.

Multiprocessor-Safe Locking Services

Locking services are used to serialize access to resources that may be used concurrently. For example, locking services can be used for insertions in a linked list, which require several pointer updates. If the update sequence by one process is interrupted by a second process that tries to access the same list, an error can occur. A sequence of operations that should not be interrupted is called a *critical section*.

Locking services use a lock word to indicate the lock status: 0 (zero) can be used for free, and 1 (one) for busy. Therefore, a service to acquire a lock would do the following:

```
test the lock word
if the lock is free
        set the lock word to busy
        return SUCCESS
...
```

Because this sequence of operations (read, test, set) is itself a critical section, special handling is required. On a uniprocessor system, disabling interrupts during the critical section prevents interruption by a context switch. But on a multiprocessor system, the hardware must provide a so-called test-and-set primitive, usually with a special machine instruction. In addition, special processor dependent synchronization

instructions called *import and export fences* are used to temporarily block other reads or writes. They protect against concurrent access by several processors and against the read and write reordering performed by modern processors.

To mask this complexity and provide independence from these machine-dependent instructions, three subroutines are defined:

_check_lock	Conditionally updates a single word variable atomically, issuing an *import fence* for multiprocessor systems. The **compare_and_swap** routine is similar, but it does not issue an import fence and, therefore, is not usable to implement a lock.
_clear_lock	Atomically writes a single word variable, issuing an *export fence* for multiprocessor systems.
_safe_fetch	Atomically reads a single word variable, issuing an *import fence* for multiprocessor systems. The import fence ensures that the read value is not a stale value resulting from an early pre-fetch. This subroutine is rarely needed.

Locking Services Example

The multiprocessor-safe locking subroutines can be used to create custom high-level routines independent of the threads library. The example that follows shows partial implementations of subroutines similar to the **pthread_mutex_lock** and **pthread_mutex_unlock** subroutines in the threads library:

```
#include <sys/atomic_op.h>       /* for locking primitives */
#define SUCCESS          0
#define FAILURE          -1
#define LOCK_FREE        0
#define LOCK_TAKEN       1

typdef struct {
        atomic_p         lock;   /* lock word */
        tid_t            owner;  /* identifies the lock owner */
        ...              /* implementation dependent fields */
} my_mutex_t;

...
int my_mutex_lock(my_mutex_t *mutex)
{
tid_t   self;   /* caller's identifier */

        /*
        Perform various checks:
          is mutex a valid pointer?
          has the mutex been initialized?
        */
        ...
        /* test that the caller does not have the mutex */
        self = thread_self();
        if (mutex->owner == self)
                return FAILURE;

        /*
        Perform a test-and-set primitive in a loop.
        In this implementation, yield the processor if failure.
        Other solutions include: spin (continuously check);
                 or yield after a fixed number of checks.
        */
        while (!_check_lock(&mutex->lock, LOCK_FREE, LOCK_TAKEN))
                yield();

        mutex->owner = self;
        return SUCCESS;
} /* end of my_mutex_lock */

int my_mutex_unlock(my_mutex_t *mutex)
{
        /*
        Perform various checks:
```

```
          is mutex a valid pointer?
          has the mutex been initialized?
        */
        ...
        /* test that the caller owns the mutex */
        if (mutex->owner != thread_self())
                return FAILURE;
        _clear_lock(&mutex->word, LOCK_FREE);
        return SUCCESS;
} /* end of my_mutex_unlock */
```

Kernel Programming

Kernel programming is thoroughly explained in *AIX 5L Version 5.1 Kernel Extensions and Device Support Programming Concepts*. This section only highlights the major changes required for multiprocessor systems.

Serialization is often required when accessing certain critical resources. Locking services can be used to serialize thread access in the process environment, but they will not protect against an access occurring in the interrupt environment. Previously, a kernel service disabled interrupts using the **i_disable** kernel service to serialize interrupt level access. However, this strategy does not work in a multiprocessor environment. Therefore, new or ported code should use the **disable_lock** and **unlock_enable** kernel services, which use simple locks in addition to interrupt control. These kernel services can also be used for uniprocessor systems, on which they simply use interrupt services without locking. See Locking Kernel Services in *AIX 5L Version 5.1 Kernel Extensions and Device Support Programming Concepts* for detailed information.

Device drivers by default run in a logical uniprocessor environment, in what is called *funnelled* mode. Most well-written drivers for uniprocessor systems will work without modification in this mode, but must be carefully examined and modified to benefit from multiprocessing. Finally, kernel services for timers now have return values because they will not always succeed in a multiprocessor environment. Therefore, new or ported code must check these return values. See Using Multiprocessor-Safe Timer Services in *AIX 5L Version 5.1 Kernel Extensions and Device Support Programming Concepts* for more information.

32-bit and 64-bit Addressability

In AIX, applications are either 32-bit applications or 64-bit applications.

A 32-bit application is an application that executes in an environment where addresses are 32 bits long and can represent 4 gigabytes of addressability (virtual address space).

A 64-bit application is an application that executes in an environment where addresses are 64 bits long and can represent much larger addressability (over a billion gigabytes).

When creating an application, a decision must be made whether to create a 32-bit application or a 64-bit application. 32-bit applications can be run on any RS/6000 system. 64-bit applications can only be run on 64-bit RS/6000 systems. Both 32-bit and 64-bit applications (and libraries) can be compiled and linked on both 32-bit and 64-bit systems.

Performance

If the same source code is used to create both a 32-bit and a 64-bit application, the 64-bit application will be the same size or larger than the 32-bit application and will generally run no faster (and often slower) than the 32-bit application, unless it makes use of the larger 64-bit addressability to improve its performance. Therefore, the correct choice should generally be to create a 32-bit application unless 64-bit addressability is required by the application or can be used to dramatically improve its performance. For this reason, the default mode for development tools is to create 32-bit objects and applications.

The 64-bit address space can be used to dramatically improve the performance of applications that manipulate large amounts of data. This data can either be created within the application or obtained from files. Generally the performance gain comes from the fact that the 64-bit application can contain the data in its address space (either created in data structures or mapped into memory), where the data would not fit into a 32-bit address space. The data would need to be multiple gigabytes in size or larger to show this benefit.

There are two reasons for this performance improvement. First, the system call overhead of reading and writing files can be avoided by mapping the files into memory. Second, 64-bit systems can support physical memories that are larger than the addressability of 32-bit applications, so 64-bit applications are needed to make full use of the physical memory available.

64-bit objects and archive file types

64-bit libraries and applications can only be created from 64-bit objects. A 64-bit object is an object type (64-bit XCOFF format) created by compilation or assembly in 64-bit mode. (This does not mean that the compiler or assembler executes in a 64-bit execution environment, just that the compiler or assembler has been requested to create 64-bit objects rather than 32-bit objects.) 32-bit XCOFF format was the only object type in releases of AIX before AIX 4.3.

There is no way to create an object or application using both 32-bit and 64-bit object files. A system provided library contains both 32-bit and 64-bit object files. The linker selects the appropriate objects from the library based on the type of linking that is requested (32-bit or 64-bit) and creates an object or application of that type.

There are two archive file types. The first does not recognize 64-bit object files and cannot be larger than 2 gigabytes. This was the only archive file type in releases of AIX before AIX 4.3. The second archive file type recognizes both 32-bit and 64-bit object files and will work with files larger than 2 gigabytes.

Differences between 32-bit and 64-bit execution environments

In addition to the differences in addressability, there are the following differences between the 32-bit and 64-bit execution environments (or modes):

- The C ″long″ type (and types derived from it) in 64-bit mode is 64 bits in size.
- All pointer types in 64-bit mode are 64 bits in size.
- 64-bit applications can make use of 64-bit PowerPC instructions.
- The size of a machine register is 64 bits in 64-bit mode.
- The maximum theoretical limits for the size of 64-bit applications, their heaps, stacks, shared libraries, and loaded objects is millions of gigabytes. The practical limits are dependent on the file system limits, paging space sizes, and system resources available.

All C fundamental types other than ″long″ and pointer types will be the same size in 32-bit and 64-bit compilation modes.

Tools support for 64-bit development

AIX provides support in all the standard tools for building, examining, and debugging 64-bit applications.

yacc, **lex**, and **lint** work with source code destined for both 32-bit and 64-bit compilation.

The C and Fortran compilers and the assembler allow the creation of both 32-bit and 64-bit objects. The linker allows the creation of both 32-bit and 64-bit objects and applications.

make, **ar**, **strip**, **dump**, **nm**, **prof**, **gprof**, **lorder**, **ranlib**, **size**, **strings**, and **sum** work with both 32-bit and 64-bit objects and applications.

dbx and **xldb** allow the debugging of both 32-bit and 64-bit applications.

Porting source code from 32-bit to 64-bit execution environments

The following issues must be examined and dealt with when porting source code for 32-bit applications to be compiled in 64-bit mode to create 64-bit applications:

- Remove any assumption that a pointer type can fit in a C integer type (or types derived from integer).
- Remove any assumption that a C long type can fit in a C integer type (or types derived from long and integer).
- Remove any assumption about the number of bits in a C long type object when bit shifting or doing bitwise operations.
- Remove any assumption that a C integer can be passed to an unprototyped long or pointer parameter.
- Remove any assumption that an unprototyped function can return a pointer or long.

The **-t** flag to **lint** can be used to find issues when porting source code from 32-bit to 64-bit compilation mode.

64-bit application development

The APIs (Application Programming Interfaces) provided to 32-bit applications are also generally provided to 64-bit applications. Some libraries that have been superseded or deprecated for 32-bit applications are not being provided to 64-bit applications, so their APIs will be missing in 64-bit execution mode.

The names of types, global variables, preprocessor macros, and predefined constants are the same in 32-bit and 64-bit compilation mode. The sizes (and layouts in the case of structures) and values for these are often different in 32-bit and 64-bit compilation mode, to account for the different sizes of the address spaces and fundamental types involved.

The names of API functions, the types of parameters passed, and the return types are the same in 32-bit and 64-bit compilation modes. The sizes (and layouts in the case of structures) of the parameters and return values are often different in 32-bit and 64-bit compilation modes to account for the different size of the address spaces involved.

64-bit library development

Libraries should provide the same functionality to 64-bit applications that they provide to 32-bit applications. This is to minimize the amount of porting that needs to be done when changing the execution mode of an application from 32-bit to 64-bit. To ease porting, the names of functions provided, their parameter types and return types should be the same for 32-bit and 64-bit applications.

The choice of the types of integral parameters and return values should be made based upon what a parameter or return value is representing. If it represents the size of an object in the address space, its type should be based upon a C ″long″ type. Otherwise, the type should be made one of the C types ″char″, ″short″, ″int″, or ″long long″, depending on what the maximum possible value is. (These types are the same size in 32-bit and 64-bit compilation mode.)

Libraries should contain both the 32-bit and 64-bit objects files for the API they support. This will minimize the porting effort for makefiles for applications that are being ported from 32-bit to 64-bit execution mode. System libraries provide object files for using both 32-bit and 64-bit applications in the same library archive file.

Only 32-bit objects can be loaded by a 32-bit application. Only 64-bit objects can be loaded by 64-bit applications. If an API is provided by loading objects, a separate 32-bit and 64-bit version of the object must be provided with a different pathname.

64-bit kernel extension development

AIX kernel extensions run in 32-bit mode on the 32-bit kernel and in 64-bit mode on the 64-bit kernel, regardless of the mode of the application for which they might be processing requests.

Kernel extensions that have not been designed to work with 64-bit applications only support 32-bit applications. A 64-bit application will fail to link attempts to make use of a system call from a kernel extention that has not been modified to support 64-bit applications. A kernel extension can indicate that it supports 64-bit applications by setting the SYS_64BIT flag when it is loaded using the **sysconfig** routine.

Kernel extension support for 64-bit applications has two aspects.

The first aspect is the use of new kernel services for working with the 64-bit user address space. The new 64-bit services for examining and manipulating the 64-bit address space are **as_att64**, **as_det64**, **as_geth64**, **as_puth64**, **as_seth64**, and **as_getsrval64**.

The new services for copying data to or from 64-bit address spaces are **copyin64**, **copyout64**, **copyinstr64**, **fubyte64**, **fuword64**, **subyte64**, and **suword64**.

The new service for doing cross-memory attaches to memory in a 64-bit address space is **xmattach64**.

The new services for creating real memory mappings are **rmmap_create64** and **rmmap_remove64**.

The major difference between all these services and their 32-bit counterparts is that they use 64-bit user addresses rather than 32-bit user addresses.

The service for determining whether a process (and its address space) is 32-bit or 64-bit is IS64U.

Performance Monitor API Programming Concepts

The following information provides an overview of the Performance Monitor API library .

Read the following to learn more about programming the Performance Monitor API for threads:

Introduction

This article describes the libpmapi library which contains a set of Application Programming Interfaces designed to provide access to some of the counting facilities of the Performance Monitor feature included in selected IBM micro-processors in the POWERPC family. Those APIs include :

- a set of system level APIs : to allow counting of the activity of a whole machine or of a set of processes with a common ancestor.
- a set of first party kernel thread level APIs : to allow threads running in 1:1 mode to count their own activity.
- a set of third party kernel thread level APIs to allow a debugger to count the activity of target threads running in 1:1 mode.

The APIs and the events available on each of the supported processors have been completely separated by design. The events available, which are different on each processor, and their descriptions as well as their current testing status are in separately installable tables, and are not described here at all because none of the API calls depend on the availability or status of any of the events.

The status of an event, as returned in bitflags by the API initialization routine **pm_init**, can be *verified, unverified,* or works with some *caveat* (see next section for an important warning ("Performance Monitor Accuracy Warning") about testing status and event accuracy).

An event filter, which is any combination of the status bits, must be passed to **pm_init** to force the return of only events with a status matching the filter. If no filter is passed to **pm_init**, no events will be returned.

For each event, in addition to a testing status and a full description, the identifier to be used in subsequent API calls, and a short and a long name are also returned by **pm_init**. The short name is a mnemonic name in the form PM_MNEMONIC. Events that are the same on different processors will have the same mnemonic name. For instance PM_CYC and PM_INST_CMPL are respectively the number of processor cycles and instruction completed and should exist on most processors.

Performance Monitor Accuracy Warning

Only events marked *verified* have gone through full verification. Events marked *caveat* have been verified within the limitations documented in the event description returned by **pm_init**.

Events marked *unverified* have undefined accuracy. Use caution with *unverified* events; the PM API is essentially providing a service to read hardware registers which may or may not have any meaningful content.

Users may experiment with *unverified* event counters and determine for themselves what, if any, use they may have for specific tuning situations.

Performance Monitor Context and State

Definitions

To provide Performance Monitor data access at various levels, support has been added to the Operating System for optional Performance Monitoring contexts. These contexts are an extension to the regular processor and thread contexts and include one 64 bit counter per hardware counter and a set of control words. The control words define what events get counted and when counting is on or off.

System level context and accumulation

For the system level APIs, optional PM contexts can be associated with each of the processors. When installed, the PM kernel extension automatically handles 32 bit PM hardware counter overflows, and maintains per-processor sets of 64 bit accumulation counters, one per 32 bit hardware PM counter.

Thread context

Optional PM contexts can also be associated with each kernel thread. The Operating System and the PM kernel extension automatically maintain sets of 64 bit counters for each of these contexts.

Thread group and process context

The concept of thread group is optionally supported by the thread level APIs. All the threads within a group, in addition to their own PM context, share a group accumulation context. A group is defined as all the threads created by a common ancestor thread. By definition, all the threads in a thread group count the same set of events, and, with one exception described below, the group must be created before any of the descendant threads are created. The second restriction stems from the fact that once descendant threads are created, it is impossible to determine a list of threads with a common ancestor. One special case of a group is the collection of all the threads belonging to a process. Such a group can be created at any time regardless of when the descendant threads are created. This is made possible by the fact that a list of threads belonging to a process can be generated. Multiple groups can coexist within a process, but

each thread can be a member of only one PM counting group. Since all the threads within a group must be counting the same events, a process group creation will fail if any thread within the process already has a context.

PM state Inheritance

The PM state is defined as the combination of the PM programmation (the events being counted), the counting state (on or off), and the optional thread group membership. There is a counting state associated with each group. When the group is created, its counting state is inherited from the initial thread in the group. For threads member of a group, the effective counting state is the result of ANDing their own counting state with the group counting state. This provides a way to effectively turn counting on and off for all threads in a group. Simply manipulating the group counting state will affect the effective counting state of all the threads in the group. Threads inherit their complete PM state from their parents when the thread is created. A thread PM context data (the value of the 64 bit counters) is not inherited, i.e. newly created threads start with counters set to zero.

PM context independence

The thread and thread group PM contexts are independent. This allows each of the thread or group of threads on a system to program themselves to be counted with their own list of events. In other words, except when using the system level API, there is no requirement that all threads counts the same events and, using a debugger, a user can certainly program threads or groups of threads to count different events.

Thread and thread group accumulation

When a thread gets suspended (or re-dispatched), its 64 bit accumulation counters are updated. If the thread is member of a group, the group accumulation counters are updated at the same time.

Similarly, when a thread stops counting or reads its PM data, its 64 bit accumulation counters are also updated by adding the current value of the PM hardware counters to them. Again, if the thread is member of a group, the group accumulation counters are also updated, regardless of whether the counter read or stop was for the thread or the thread group.

The group level accumulation data is kept consistent with the individual PM data for the thread members of the group, whenever possible. When a thread voluntarily leaves a group, i.e., deletes its PM context, its accumulated data is automatically subtracted from the group level accumulated data. Similarly, when a thread member in a group resets its own data, the data in question is subtracted from the group level accumulated data. Note that when a thread dies, no action is taken on the group accumulated data.

The only situation where the group level accumulation is not consistent with the sum of the data for each of its members is when the group level accumulated data has been reset, and the group has more then one member. This situation is detected and marked by a bit returned when the group data is read.

Security Considerations

- System level security

  The system level APIs calls are only available from the super user except when the process tree option is used. In that case a locking mechanism prevents calls to be made from more than one process. This mechanism ensures ownership of the API and exclusive access by one process from the time the system level contexts are created until they are deleted.

  Turning on the process tree option results in counting for only the calling process and its descendants; the default is to count all activities on each processor.
- Thread and thread group level security

Since the system level APIs would report bogus data if thread contexts where in use, it is not allowed to make system level API calls at the same time as thread level API calls. The allocation of the first thread context will take the system level API lock which will not be released until the last context has been deallocated.

When using first party calls, a thread is only allowed to modify its own PM context. The only exception to this rule is when making group level calls, which obviously affect the group context, but can also affect other threads context. Indeed, deleting a group deletes all the contexts associated with the group, i.e., the caller context, the group context and all the contexts belonging to all the threads in the group.

Access to a PM context not belonging to the calling thread or its group is only available from the target process's debugger. The third party API calls only succeed when the target process is being ptraced by the API caller, i.e., the caller is already attached to the target process, and the debuggee is stopped.

The fact that the debugger must already have been attached to the debugged thread before any third party call to the API can be made, ensures that the security level of the API will be the same as the one used between debuggers and debuggees.

Common Definitions

Common rules

pm_init must be called before any other API call can be made, and only events returned by a given **pm_init** call with its associated filter setting can be used in subsequent **pm_set_program calls**. **pm_init** also returns the processor name, the number of counters available (2, 4 or 8), and the threshold multiplier. For each event returned, a thresholdable flag is also returned. This flag indicates whether an event can be used with a threshold. If so, then specifying a threshold defers counting until the threshold multiplied by the processor's threshold multiplier has been exceeded.

PM contexts cannot be reprogrammed or reused at any time. This means that none of the APIs support more than one call to a **pm_set_program** interface without a call to a **pm_delete_program** interface. This also means that when creating a process group, none of the threads in the process is allowed to already have a context.

All the API calls return 0 when successful or a positive error code (which can be decoded using **pm_error**) otherwise.

Group information

The following information is associated with each thread group :

- member count :

  the number of threads member of the group. This includes deceased threads which were member of the group when running.

  If the consistency flag is on, it will be the number of threads that have contributed to the group level data.
- process flag :

  indicates that the group includes all the threads in the process.
- consistency flag :

  indicates that the group PM data is consistent with the sum of the individual PM data for the thread members.

This information is returned by the **pm_get_data_mygroup** and **pm_get_data_group** interfaces in a **pm_groupinfo_t** structure.

The Seven Basic API Calls

Each of the seven section below describes a system-wide API call that has variations for first-party kernel thread or group counting, and third-party kernel thread or group counting. Variations are indicated by suffixes to the function call names, such as **pm_set_program**, **pm_set_program_mythread**, **pm_set_program_group** etc.

pm_set_program
: Sets the counting configuration. Use this call to specify the events to be counted, and a mode in which to count. The list of events to choose from is returned by **pm_init**. If the list includes a thresholdable event, a threshold can also be specified.

  The mode in which to count, can include user-mode and/or kernel-mode counting, and whether to start counting immediately. For the system-wide API call, the mode also include whether to turn counting on only for a process and its descendants or for the whole system. For counting group API calls, the mode includes the type of counting group to create, i.e., a group containing the initial thread and its future descendants, or a process level group, which includes all the threads in a process.

pm_get_program
: Retrieves the current Performance Monitor settings. This includes mode information and the list of events being counted. If the list includes a thresholdable event, a threshold will also be returned.

pm_delete_program
: Deletes the Performance Monitor configuration. Use this call to undo **pm_set_program**.

pm_start
: Starts Performance Monitor counting.

pm_stop
: Stops Performance Monitor counting.

pm_get_data
: Returns Performance Monitor counting data. The data is a set of 64-bit values, one per hardware counter. For the counting group API calls, the group information previously described is also returned.

pm_reset_data
: Resets Performance Monitor counting data. All values are set to 0.

Examples

Example code is also shipped to the */usr/samples/pmapi* directory.

Simple single threaded program

```
#include main()
{
       pm_info_t pminfo;
       pm_prog_t prog;
       pm_data_t data;
       int filter = PM_VERIFIED; /* use only verified events */

       pm_init(filter, &pminfo)

       prog.mode.w       = 0;  /* start with clean mode */
       prog.mode.b.user  = 1;  /* count only user mode */

       for (i = 0; i < pminfo.maxpmcs; i++)
                prog.events[i] = COUNT_NOTHING;

       prog.events[0]    = 1;  /* count event 1 in first counter */
       prog.events[1]    = 2;  /* count event 2 in second counter */

       pm_program_mythread(&prog);
```

```
       pm_start_mythread();

(1)    ... usefull work ....

       pm_stop_mythread();
       pm_get_data_mythread(&data);

       ... print results ...
}
```

Debugger example for previous program

To look at the PM data while the program is executing :

```
from a debugger at breakpoint (1)

       pm_init(filter);
(2)    pm_get_program_thread(pid, tid, &prog);
       ... display PM programmation ...

(3)    pm_get_data_thread(pid, tid);
       ... display PM data ...

       pm_delete_program_thread(pid, tid);
       prog.events[0] = 2; /* change counter 1 to count event number 2 */
       pm_set_program_thread(pid, tid, &prog);

continue program
```

The scenario above would work as well if the program being executed under the debugger didn't have any embedded PM API calls. The only difference would be that the calls at (2) and (3) would fail, and that when the program continues, it will be counting only event number 2 in counter 1, and nothing in other counters.

Simple multithreaded example

A simple multithreaded example with independent threads counting the same set of events.

```
#include pm_data_t data2;

void *
doit(void *)
{

(1)    pm_start_mythread();

       ... usefull work ....

       pm_stop_mythread();
       pm_get_data_mythread(&data2);
}

main()
{
       pthread_t threadid;
       pthread_attr_t attr;
       pthread_addr_t status;

       ... same initialization as in previous example ...

       pm_program_mythread(&prog);

       /* setup 1:1 mode, M:N not supported by APIs */
       pthread_attr_init(&attr);
       pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
       pthread_create(&threadid, &attr, doit, NULL);
```

```
(2)     pm_start_mythread();

        ... usefull work ....

        pm_stop_mythread();
        pm_get_data_mythread(&data);

        ... print main thread results (data )...

        pthread_join(threadid, &status);

        ... print auxiliary thread results (data2) ...
}
```

Counting starts at (1) and (2) for the main and auxiliary threads respectively because the initial counting state was off and it was inherited by the auxiliary thread from its creator.

Simple thread counting group example

Example with two threads in a counting group. The body of the auxiliary thread's initialization routine is the same as in the previous example.

```
main()
{
        ... same initialization as in previous example ...

        pm_program_mygroup(&prog); /* create counting group */
(1)     pm_start_mygroup()

        pthread_create(&threadid, &attr, doit, NULL)

(2)     pm_start_mythread();

        ... usefull work ....

        pm_stop_mythread();
        pm_get_data_mythread(&data)


        ... print main thread results ...

        pthread_join(threadid, &status);

        ... print auxiliary thread results ...

        pm_get_data_mygroup(&data)


        ... print group results ...
}
```

The call in (2) is necessary because the call in (1) only turns on counting for the group, not the individual threads in it. At the end, the group results are the sum of both threads results.

Thread counting example with reset

This example with a reset call illustrates the impact on the group data. The body of the auxiliary thread is the same as before, except for the **pm_start_mythread()** call which is not necessary in this case.

```
main()
{
        ... same initialization as in previous example...

        prog.mode.b.count  = 1;  /* start counting immediately */
        pm_program_mygroup(&prog);

        pthread_create(&threadid, pthread_attr_default, doit, NULL)
```

```
        ... usefull work ....

        pm_stop_mythread()
        pm_reset_data_mythread()

        pthread_join(threadid, &status);

        ...print auxiliary thread results...

        pm_get_data_mygroup(&data)


        ...print group results...
}
```

The main thread and the group counting state are both on before the auxiliary thread is created so the auxiliary thread will inherit that state and start counting immediately.

At the end, **data1** is equal to **data** because the **pm_reset_data_mythread** automatically subtracted the main thread data from the group data to keep it consistent. In fact at all time the group data is still equal to the sum of the auxiliary and the main thread data, but in this case the main thread data is null.

Related Information

pmapi.h File

Chapter 11. Threads Programming Guidelines

The following information provides guidelines for writing multi-threaded programs using the threads library (**libpthreads.a**). The AIX threads library is based on the emerging POSIX 1003.1c standard. For this reason, the following information presents the threads library as the AIX implementation of the POSIX standard.

If you want to learn how to write programs using multiple threads, you should read the topics in sequential order. If you are looking for specific information, choose one of the following topics:

Thread Implementation Model

At the other end of the spectrum is the ″kernel-thread model.″ In this model, all threads are visible to the operating system kernel. Thus, all threads are kernel scheduled entities, and all threads can concurrently execute. The threads are scheduled onto processors by the kernel according to the scheduling attributes of the threads. This model is the model provided in AIX 4.2.

AIX 4.3 uses a hybrid model that offers the speed of library threads and the concurrency of kernel threads. In hybrid models, a process has a varying number of kernel scheduled entities associated with it. It also has a potentially much larger number of library threads associated with it. Some library threads may be bound to kernel scheduled entities, while the other library threads are multiplexed onto the remaining kernel scheduled entities. For this reason, a hybrid model is referred to as a ″M:N″ model. In this model, the process can have multiple concurrently executing threads; specifically, it can have as many concurrently executing threads as it has kernel scheduled entities.

Thread-safe and Threaded Libraries in AIX

In AIX 4.2, special versions of selected libraries were provided, that were for use by threaded applications. These libraries were counterparts of the non-thread-safe libraries, but with the suffix ″_r″ added to the name. These libraries were:

libc.a/libc_r.a
libm.a/libm_r.a
libs.a/libs_r.a
libsvid.a/libsvid_r.a
libxti.a/libxti_r.a
libbsd.a/libbsd_r.a
libnetsvc.a/libnetsvc_r.a
libs2.a/libs2_r.a
libtli.a/libtli_r.a

In AIX 4.3, the need for these ″_r″ versions has been eliminated. By default, all applications are now considered ″threaded,″ even though most are of the case ″single threaded.″ These thread-safe libraries are now:

libbsd.a libc.a libm.a

libsvid.a libtli.a libxti.a
libnetsvc.a

The ″_r″ versions have been kept as links to these libraries, to enable compatibility with user applications.

Threads Versions On AIX

In order to bring threaded application support to our users, AIX introduced threads API models based on preliminary drafts of the now-official IEE POSIX standard. AIX 4.3 conforms fully to the IEEE POSIX standard for threads APIs, IEEE POSIX 1003.1-1996.

Note: In AIX 4.2 threads were supported at a ″Draft 4″ level.

AIX 4.3 provides full support for applications compiled on AIX 4.2. It also provides compilation support for applications written to the ″Draft 7″ level that are not able to modify their source code to full standard conformance.

Compiling a Threaded Application

In AIX 4.2, ″_r″ versions of the C compiler invocations were offered that allowed the proper libraries and command line options to be set for creating a threaded application.

In AIX 4.3 the use of the ″_r″ invocations is no longer required for creating a threaded application.

- To Compile a Threaded Application on AIX 4.3, use either the normal or ″_r″ version of the compiler.
- To Compile a Threaded Application at ″Draft 7″ level, use the ″_r7″ invocation of the compiler.

Threads Basic Operation Overview

To write a multi-threaded program, it is necessary to understand how to create and terminate threads. Synchronization facilities and scheduling control are not required for a basic usage of threads.

The following information will help you in writing your first multi-threaded program:

- “Creating Threads”
- “Terminating Threads” on page 219
- “List of Threads Basic Operation Subroutines” on page 226

Creating Threads

A thread has attributes, which specify the characteristics of the thread. The attributes default values fit for most common cases. To control thread attributes, a thread attributes object must be defined before creating the thread.

Thread Attributes Object

The thread attributes are stored in an opaque object, the *thread attributes object*, used when creating the thread. It contains several attributes, depending on the implementation of POSIX options. It is accessed through a variable of type **pthread_attr_t**. In AIX, the **pthread_attr_t** data type is a pointer to a structure; on other systems it may be a structure or another data type.

Thread Attributes Object Creation and Destruction

The thread attributes object is initialized to default values by the **pthread_attr_init** subroutine. The attributes are handled by subroutines. The thread attributes object is destroyed by the **pthread_attr_destroy** subroutine. This subroutine may free storage dynamically allocated by the **pthread_attr_init** subroutine, depending on the implementation of the threads library.

In the following example, a thread attributes object is created and initialized with default values, then used and finally destroyed:

```
pthread_attr_t attributes;
                /* the attributes object is created */
...
if (!pthread_attr_init(&attributes)) {
                /* the attributes object is initialized */
        ...
                /* using the attributes object */
        ...
        pthread_attr_destroy(&attributes);
                /* the attributes object is destroyed */
}
```

The same attributes object can be used to create several threads. It can also be modified between two thread creations. When the threads are created, the attributes object can be destroyed without affecting the threads created with it.

Detachstate Attribute

The following attribute is always defined.

Detachstate Specifies the detached state of a thread.

The value of the attribute is returned by the **pthread_attr_getdetachstate** subroutine; it can be set by the **pthread_attr_setdetachstate** subroutine. Possible values for this attributes are the following symbolic constants:

PTHREAD_CREATE_DETACHED Specifies that the thread will be created in the detached state.
PTHREAD_CREATE_JOINABLE Specifies that the thread will be created in the joinable state.

The default value is **PTHREAD_CREATE_JOINABLE**.

If you create a thread in the joinable state, you must pthread_join (“Calling the pthread_join Subroutine” on page 237) with the thread. Otherwise, you may run out of storage space when creating new threads, because each thread takes up a signficant amount of memory.

Other Attributes

The following attributes are also defined in AIX. They are intended for advanced programs and may require special execution privilege to take effect. Most programs will operate correctly with the default settings.

Contention Scope Specifies the contention scope of a thread.
Inheritsched Specifies the inheritance of scheduling properties of a thread.
Schedparam Specifies the scheduling parameters of a thread.
Schedpolicy Specifies the scheduling policy of a thread.

The use of these attributes is explained in Scheduling Attributes.

Stacksize Specifies the size of the thread's stack.
Stackaddr Specifies the address of the thread's stack.
Guardsize Specifies the size of the guard area of the thread's stack.

The use of these attributes is explained in "Stack Attributes" on page 251.

Thread Creation

Creating a thread is accomplished by calling the **pthread_create** subroutine. This subroutine creates a new thread and makes it runnable.

Using the Thread Attributes Object

When calling the **pthread_create** subroutine, you may specify a thread attributes object. If you specify a **NULL** pointer, the created thread will have the default attributes. Thus, the code fragment:

```
pthread_t thread;
pthread_attr_t attr;
...
pthread_attr_init(&attr);
pthread_create(&thread, &attr, init_routine, NULL);
pthread_attr_destroy(&attr);
```

is equivalent to:

```
pthread_t thread;
...
pthread_create(&thread, NULL, init_routine, NULL);
```

Entry Point Routine

When calling the **pthread_create** subroutine, you must specify an entry-point routine. This routine, provided by your program, is similar to the **main** routine for the process. It is the first user routine executed by the new thread. When the thread returns from this routine, the thread is automatically terminated.

The entry-point routine has one parameter, a void pointer, specified when calling the **pthread_create** subroutine. You may specify a pointer to some data, such as a string or a structure. The creating thread (the one calling the **pthread_create** subroutine) and the created thread must agree upon the actual type of this pointer.

The entry-point routine returns a void pointer. After the thread termination, this pointer is stored by the threads library unless the thread is detached. See "Synchronization Overview" on page 227 for more information about using this pointer.

Returned Information

The **pthread_create** subroutine returns the thread ID of the new thread. The caller can use this thread ID to perform various operations on the thread.

Depending on the scheduling parameters of both threads, the new thread may start running before the call to the **pthread_create** subroutine returns. It may even happen that, when the **pthread_create** subroutine returns, the new thread has already terminated. The thread ID returned by the **pthread_create** subroutine through the *thread* parameter is then already invalid. It is, therefore, important to check for the **ESRCH** error code returned by threads library subroutines using a thread ID as a parameter.

If the **pthread_create** subroutine is unsuccessful, no new thread is created, the thread ID in the *thread* parameter is invalid, and the appropriate error code is returned.

Handling Thread IDs

The thread ID of a newly created thread is returned to the creating thread through the *thread* parameter. The current thread ID is returned by the **pthread_self** subroutine.

A thread ID is an opaque object; its type is **pthread_t**. In AIX, the **pthread_t** data type is an integer. On other systems, it may be a structure, a pointer, or any other data type.

To enhance the portability of programs using the threads library, the thread ID should always be handled as an opaque object. For this reason, thread IDs should be compared using the **pthread_equal** subroutine. Never use the C equality operator (**==**), because the **pthread_t** data type may be neither an arithmetic type nor a pointer.

A First Multi-Threaded Program

The first multi-threaded program discussed is short. It displays ″Hello!″ in both English and French for five seconds. Compile with **cc_r** or **xlc_r**. See “Developing Multi-Threaded Programs” on page 173 for more information on compiling thread programs.

```
#include <pthread.h>    /* include file for pthreads - the 1st */
#include <stdio.h>      /* include file for printf()           */
#include <unistd.h>     /* include file for sleep()            */

void *Thread(void *string)
{
        while (1)
                printf("%s\n", (char *)string);
        pthread_exit(NULL);
}

int main()
{
        char *e_str = "Hello!";
        char *f_str = "Bonjour !";

        pthread_t e_th;
        pthread_t f_th;

        int rc;

        rc = pthread_create(&e_th, NULL, Thread, (void *)e_str);
        if (rc)
                exit(-1);
        rc = pthread_create(&f_th, NULL, Thread, (void *)f_str);
        if (rc)
                exit(-1);
        sleep(5);

        /* usually the exit subroutine should not be used
           see below to get more information */
        exit(0);
}
```

The initial thread (executing the **main** routine) creates two threads. Both threads have the same entry-point routine (the **Thread** routine), but a different parameter. The parameter is a pointer to the string that will be displayed.

Terminating Threads

A thread automatically terminates when it returns from its entry-point routine. A thread can also explicitly terminate itself or terminate any other thread in the process. Because all threads share the same data space, a thread must perform cleanup operations at termination time; cleanup handlers are provided by the threads library for this purpose.

Exiting a Thread

A process can exit at any time by any thread by calling the **exit** subroutine. Similarly, a thread can exit at any time by calling the **pthread_exit** subroutine.

Calling the **exit** subroutine terminates the entire process, including all its threads. In a multi-threaded program, the **exit** subroutine should only be used when the entire process needs to be terminated; for example, in the case of an unrecoverable error. The **pthread_exit** subroutine should be preferred, even for exiting the initial thread.

Calling the **pthread_exit** subroutine terminates the calling thread. The *status* parameter is saved by the library and can be further used when joining ("Joining Threads" on page 236) the terminated thread .
Calling the **pthread_exit** subroutine is similar, but not identical, to returning from the thread's initial routine. The result of returning from the thread's initial routine depends on the thread:

- Returning from the initial thread implicitly calls the **exit** subroutine, thus terminating all the threads in the process.
- Returning from another thread implicitly calls the **pthread_exit** subroutine. The return value has the same role as the *status* parameter of the **pthread_exit** subroutine.

It is recommended always to use the **pthread_exit** subroutine to exit a thread to avoid implicitly calling the **exit** subroutine.

Exiting the initial thread (for example by calling the **pthread_exit** subroutine from the **main** routine) does not terminate the process. It only terminates the initial thread. If the initial thread is terminated, the process will be terminated when the last thread in it terminates. In this case, the process return code (usually the return value of the **main** routine or the parameter of the **exit** subroutine) is 0 if the last thread was detached or 1 otherwise.

The following example is a slightly modified version of our first multi-threaded program. The program displays exactly 10 messages in each language. This is accomplished by calling the **pthread_exit** subroutine in the **main** routine after creating the two threads, and creating a loop in the **Thread** routine.

```
#include <pthread.h>    /* include file for pthreads - the 1st */
#include <stdio.h>      /* include file for printf()           */

void *Thread(void *string)
{
        int i;

        for (i=0; i<10; i++)
                printf("%s\n", (char *)string);
        pthread_exit(NULL);
}

int main()
{
        char *e_str = "Hello!";
        char *f_str = "Bonjour !";

        pthread_t e_th;
        pthread_t f_th;

        int rc;

        rc = pthread_create(&e_th, NULL, Thread, (void *)e_str);
        if (rc)
                exit(-1);
        rc = pthread_create(&f_th, NULL, Thread, (void *)f_str);
        if (rc)
                exit(-1);
        pthread_exit(NULL);
}
```

It is important to note that the **pthread_exit** subroutine frees any thread-specific data, including the thread's stack. Any data allocated on the stack becomes invalid, since the stack is freed and the corresponding memory may be reused by another thread. Therefore, thread synchronization objects (mutexes and condition variables) allocated on a thread's stack must be destroyed before the thread calls the **pthread_exit** subroutine.

Unlike the **exit** subroutine, the **pthread_exit** subroutine does not clean up system resources shared among threads. For example, files are not closed by the **pthread_exit** subroutine, since they may be used by other threads.

Canceling a Thread

The thread cancellation mechanism allows a thread to terminate the execution of any other thread in the process in a controlled manner. The target thread (that is, the one that's being canceled) can hold cancellation requests pending in a number of ways and perform application-specific cleanup processing when the notice of cancellation is acted upon. When canceled, the thread implicitly calls the **pthread_exit((void *)-1)** subroutine.

The cancellation of a thread is requested by calling the **pthread_cancel** subroutine. When the call returns, the request has been registered, but the thread may still be running. The call to the **pthread_cancel** subroutine is unsuccessful only when the specified thread ID is not valid.

Cancelability State and Type

The cancelability state and type of a thread determines the action taken upon receipt of a cancellation request. Each thread controls its own cancelability state and type with the **pthread_setcancelstate** and **pthread_setcanceltype** subroutines.

There are two possible cancelability states and two possible cancelability types, leading to three possible cases, as shown in the following table.

Cancelability State	Cancelability Type	Resulting Case
Disabled	Any (the type is ignored)	Case 1
Enabled	Deferred	Case 2
Enabled	Asynchronous	Case 3

The following discusses the three possible cases.

1. *Disabled cancelability*. Any cancellation request is set pending, until the cancelability state is changed or the thread is terminated in another way.

   A thread should disable cancelability only when performing operations that cannot be interrupted. For example, if a thread is performing some complex file save operations (such as an indexed database) and is canceled during the operation, the files may be left in an inconsistent state. To avoid this, the thread should disable cancelability during the file save operations.
2. *Deferred cancelability*. Any cancellation request is set pending until the thread reaches the next cancellation point. It is the default cancelability state.

   This cancelability state ensures that a thread can be cancelled, but limits the cancellation to specific moments in the thread's execution, called *cancellation points*. A thread canceled on a cancellation point leaves the system in a safe state; however, user data may be inconsistent or locks may be held by the canceled thread. To avoid these situations, you may use cleanup handlers or disable cancelability within critical regions. See "Using Cleanup Handlers" on page 225 for more information about cleanup handlers.
3. *Asynchronous cancelability*. Any cancellation request is acted upon immediately.

A thread that is asynchronously canceled while holding resources may leave the process, or even the system, in a state from which it is difficult or impossible to recover. See “Async-Cancel Safety” for more information about async-cancel safety.

Async-Cancel Safety

A function is said to be *async-cancel safe* if it is written so that calling the function with asynchronous cancelability enabled does not cause any resource to be corrupted, even if a cancellation request is delivered at any arbitrary instruction.

Any function that gets a resource as a side effect cannot be made async-cancel safe. For example, if the **malloc** subroutine is called with asynchronous cancelability enabled, it might acquire the resource successfully, but as it was returning to the caller, it could act on a cancellation request. In such a case, the program would have no way of knowing whether the resource was acquired or not.

For this reason, most library routines cannot be considered async-cancel safe. It is recommended not to use asynchronous cancelability unless you are sure only to perform operations that do not hold resources and only to call async-cancel safe library routines.

The following three subroutines are async-cancel safe; they ensure that cancellation will be properly handled, even if asynchronous cancelability is enabled:

- **pthread_cancel**
- **pthread_setcancelstate**
- **pthread_setcanceltype**

An alternative to asynchronous cancelability is to use deferred cancelability and to add explicit cancellation points by calling the **pthread_testcancel** subroutine (see “Cancellation Points” for more information).

Cancellation Points

Cancellation points are points inside of certain subroutines where a thread must act on any pending cancellation request if deferred cancelability is enabled. All these subroutines may block the calling thread or compute indefinitely.

An explicit cancellation point can also be created by calling the **pthread_testcancel** subroutine. This subroutine simply creates a cancellation point. If deferred cancelability is enabled, and if a cancellation request is pending, the request is acted upon and the thread is terminated. Otherwise, the subroutine simply returns.

Other cancellation points occur when calling the following subroutines:

- **pthread_cond_wait**
- **pthread_cond_timedwait**
- **pthread_join**

The **pthread_mutex_lock** and **pthread_mutex_trylock** subroutines do not provide a cancellation point. If they did, all functions calling these subroutines (and many functions do) would provide a cancellation point. Having too many cancellation points makes programming very difficult, requiring either lots of disabling and restoring of cancelability or extra effort in trying to arrange for reliable cleanup at every possible place. See “Using Mutexes” on page 227 for more information about these subroutines.

Cancellation Points

Cancellation points occur when a thread is executing the following functions:

aio_suspend	close

creat
fcntl
fsync
getmsg
getpmsg
lockf
mq_receive
mq_send
msgrcv
msgsnd
msync
nanosleep
open
pause
poll
pread
pthread_cond_timedwait
pthread_cond_wait
pthread_join
pthread_testcancel
putpmsg
pwrite
read
readv
select
sem_wait
sigpause
sigsuspend
sigtimedwait
sigwait
sigwaitinfo
sleep
system
tcdrain
usleep
wait
wait3
waitid
waitpid
write
writev

A cancellation point may also occur when a thread is executing the following functions:

catclose
catgets
catopen
closedir
closelog
ctermid
dbm_close
dbm_delete
dbm_fetch
dbm_nextkey
dbm_open
dbm_store
dlclose
dlopen
endgrent
endpwent
fwprintf
fwrite
fwscanf
getc
getc_unlocked
getchar
getchar_unlocked
getcwd
getdate
getgrent
getgrgid
getgrgid_r
getgrnam
getgrnam_r
getlogin
getlogin_r
popen
printf
putc
putc_unlocked
putchar
putchar_unlocked
puts
pututxline
putw
putwc
putwchar
readdir
readdir_r
remove
rename
rewind
endutxent
fclose
fcntl
fflush
fgetc
fgetpos
fgets
fgetwc
fgetws
fopen
fprintf
fputc
fputs
getpwent
getpwnam
getpwnam_r
getpwuid
getpwuid_r
gets
getutxent
getutxid
getutxline
getw
getwc
getwchar
getwd
rewinddir
scanf
seekdir
semop
setgrent
setpwent
setutxent
strerror
syslog
tmpfile
tmpnam
ttyname
ttyname_r
fputwc
fputws
fread
freopen
fscanf
fseek

fseeko	fsetpos	ftell
ftello	ftw	glob
iconv_close	iconv_open	ioctl
lseek	mkstemp	nftw
opendir	openlog	pclose
perror	ungetc	ungetwc
unlink	vfprintf	vfwprintf
vprintf	vwprintf	wprintf
wscanf		

The side effects of acting upon a cancellation request while suspended during a call of a function is the same as the side effects that may be seen in a single-threaded program when a call to a function is interrupted by a signal and the given function returns [EINTR]. Any such side effects occur before any cancellation cleanup handlers are called.

Whenever a thread has cancelability enabled and a cancellation request has been made with that thread as the target and the thread calls pthread_testcancel, then the cancellation request is acted upon before pthread_testcancel returns. If a thread has cancelability enabled and the thread has an asynchronous cancellation request pending and the thread is suspended at a cancellation point waiting for an event to occur, then the cancellation request will be acted upon. However, if the thread is suspended at a cancellation point and the event that it is waiting for occurs before the cancellation request is acted upon, it is dependent upon the sequence of events whether the cancellation request is acted upon or whether the request remains pending and the thread resumes normal execution.

Cancellation Example

The following example is a variant of our first multi-threaded program. Both ″writer″ threads are canceled after 10 seconds, and after they have written their message at least 5 times.

```
#include <pthread.h>    /* include file for pthreads - the 1st */
#include <stdio.h>      /* include file for printf()           */
#include <unistd.h>     /* include file for sleep()            */

void *Thread(void *string)
{
        int i;
        int o_state;

        /* disables cancelability */
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &o_state);

        /* writes five messages */
        for (i=0; i<5; i++)
                printf("%s\n", (char *)string);

        /* restores cancelability */
        pthread_setcancelstate(o_state, &o_state);

        /* writes further */
        while (1)
                printf("%s\n", (char *)string);
        pthread_exit(NULL);
}

int main()
{
        char *e_str = "Hello!";
        char *f_str = "Bonjour !";

        pthread_t e_th;
        pthread_t f_th;
```

```
        int rc;

        /* creates both threads */
        rc = pthread_create(&e_th, NULL, Thread, (void *)e_str);
        if (rc)
                return -1;
        rc = pthread_create(&f_th, NULL, Thread, (void *)f_str);
        if (rc)
                return -1;


        /* sleeps a while */
        sleep(10);

        /* requests cancellation */
        pthread_cancel(e_th);
        pthread_cancel(f_th);

        /* sleeps a bit more */
        sleep(10);
        pthread_exit(NULL);
}
```

Using Cleanup Handlers

Cleanup handlers provide an easy way to implement a portable mechanism for releasing resources and restoring invariants when a thread terminates.

Calling Cleanup Handlers

Cleanup handlers are specific to each thread. A thread can have several cleanup handlers; cleanup handlers are stored in a thread-specific LIFO stack. They are all called in the following cases:

- The thread returns from its entry-point routine.
- The thread calls the **pthread_exit** subroutine.
- The thread acts on a cancellation request.

A cleanup handler is pushed onto the cleanup stack, by the **pthread_cleanup_push** subroutine. The **pthread_cleanup_pop** subroutine pops the topmost cleanup handler from the stack, and optionally executes it. Use this subroutine when the cleanup handler is no longer needed.

The cleanup handler is a user-defined routine. It has one parameter, a void pointer, specified when calling the **pthread_cleanup_push** subroutine. You may specify a pointer to some data the cleanup handler needs to perform its operation.

In the following example, a buffer is allocated for performing some operation. With deferred cancelability enabled, the operation may be stopped at any cancellation point. A cleanup handler is established to free the buffer in that case.

```
/* the cleanup handler */

cleaner(void *buffer)

{
        free(buffer);
}
/* fragment of another routine */
...
myBuf = malloc(1000);
if (myBuf != NULL) {

        pthread_cleanup_push(cleaner, myBuf);
```

```
        /*
         *       perform any operation using the buffer,
         *       including calls to other functions
         *       and cancellation points
         */

        /* pops the handler and frees the buffer in one call */
        pthread_cleanup_pop(1);
}
```

Using deferred cancelability ensures that the thread will not act on any cancellation request between the buffer allocation and the registration of the cleanup handler, because neither the **malloc** subroutine nor the **pthread_cleanup_push** subroutine provides any cancellation point. When popping the cleanup handler, the handler is executed, freeing the buffer. More complex programs may not execute the handler when popping it, because the cleanup handler should be thought of as an emergency exit for the protected portion of code.

Balancing the Push and Pop Operations

The **pthread_cleanup_push** and **pthread_cleanup_pop** subroutines should always appear in pairs within the same lexical scope, that is, within the same function and the same statement block. They can be thought of as left and right parentheses enclosing a protected portion of code.

The reason for this rule is that on some systems these subroutines are implemented as macros. The **pthread_cleanup_push** subroutine is implemented as a left brace, followed by other statements:

```
#define pthread_cleanup_push(rtm,arg) { \
         /* other statements */
```

The **pthread_cleanup_pop** subroutine is implemented as a right brace following other statements:

```
#define pthread_cleanup_pop(ex) \
         /* other statements */  \
}
```

Not following the balancing rule for the **pthread_cleanup_push** and **pthread_cleanup_pop** subroutines may lead to compiler errors or to unexpected behavior of your programs when porting to other systems.

In AIX, the **pthread_cleanup_push** and **pthread_cleanup_pop** subroutines are library routines, and can be unbalanced within the same statement block. However, they must be balanced in the program, since the cleanup handlers are stacked.

List of Threads Basic Operation Subroutines

pthread_attr_destroy	Deletes a thread attributes object.
pthread_attr_getdetachstate	Returns the value of the detachstate attribute of a thread attributes object.
pthread_attr_init	Creates a thread attributes object and initializes it with default values.
pthread_create	Creates a new thread, initializes its attributes, and makes it runnable.
pthread_cancel	Requests the cancellation of a thread.
pthread_cleanup_pop	Removes, and optionally executes, the routine at the top of the calling thread's cleanup stack.
pthread_cleanup_push	Pushes a routine onto the calling thread's cleanup stack.
pthread_equal	Compares two thread IDs.
pthread_exit	Terminates the calling thread.
pthread_self	Returns the calling thread's ID.
pthread_setcancelstate	Sets the calling thread's cancelability state.
pthread_setcanceltype	Sets the calling thread's cancelability type.

pthread_testcancel Creates a cancellation point in the calling thread.

Synchronization Overview

One main benefit of using threads is the ease of using synchronization facilities. Three basic synchronization techniques are implemented in the threads library: mutexes, condition variables, and joining. More complex synchronization objects can be built using the primitive objects. This is discussed in “Making Complex Synchronization Objects” on page 252.

Using Mutexes

A mutex is a mutual exclusion lock. Only one thread can hold the lock. Mutexes are used to protect data or other resources from concurrent access. A mutex has attributes, which specify the characteristics of the mutex. In the current version of AIX, the mutex attributes are not used. The mutex attributes object can therefore be ignored when creating a mutex.

Mutex Attributes Object

Like threads, mutexes are created with the help of an attributes object. The mutex attributes object is an abstract object, containing several attributes, depending on the implementation of POSIX options. It is accessed through a variable of type **pthread_mutexattr_t**. In AIX, the **pthread_mutexattr_t** data type is a pointer; on other systems, it may be a structure or another data type.

Mutex Attributes Object Creation and Destruction

The mutex attributes object is initialized to default values by the **pthread_mutexattr_init** subroutine. The attributes are handled by subroutines. The thread attributes object is destroyed by the **pthread_mutexattr_destroy** subroutine. This subroutine may free storage dynamically allocated by the **pthread_mutexattr_init** subroutine, depending on the implementation of the threads library.

In the following example, a mutex attributes object is created and initialized with default values, then used and finally destroyed:

```
pthread_mutexattr_t attributes;
                /* the attributes object is created */
...
if (!pthread_mutexattr_init(&attributes)) {
                /* the attributes object is initialized */
        ...
                /* using the attributes object */
        ...
        pthread_mutexattr_destroy(&attributes);
                /* the attributes object is destroyed */
}
```

The same attributes object can be used to create several mutexes. It can also be modified between two mutex creations. When the mutexes are created, the attributes object can be destroyed without affecting the mutexes created with it.

Mutex Attributes

In AIX, no mutex attribute is defined. They depend on POSIX options (“Threads Library Options” on page 261) that are not implemented in AIX . However, the following attributes may be defined on other systems:

Protocol Specifies the protocol used to prevent priority inversions for a mutex. This attribute depends on either the priority inheritance or the priority protection POSIX option (“Threads Library Options” on page 261).

Prioceiling	Specifies the priority ceiling of a mutex. This attribute depends on the priority protection POSIX option (“Threads Library Options” on page 261).
Process-shared	Specifies the process sharing of a mutex. This attribute depends on the process sharing POSIX option (“Threads Library Options” on page 261).

The default values for these attributes are sufficient for most simple cases. See “Synchronization Scheduling” on page 243 for more information about the protocol and prioceiling attributes; see “Advanced Attributes” on page 250 for more information about the process-shared attribute.

Creating and Destroying Mutexes

A mutex is created by calling the **pthread_mutex_init** subroutine. You may specify a mutex attributes object. If you specify a **NULL** pointer, the mutex will have the default attributes. Thus, the code fragment:

```
pthread_mutex_t mutex;
pthread_mutex_attr_t attr;
...
pthread_mutexattr_init(&attr);
pthread_mutex_init(&mutex, &attr);
pthread_mutexattr_destroy(&attr);
```

is equivalent to:

```
pthread_mutex_t mutex;
...
pthread_mutex_init(&mutex, NULL);
```

The ID of the created mutex is returned to the calling thread through the *mutex* parameter. The mutex ID is an opaque object; its type is **pthread_mutex_t**. In AIX, the **pthread_mutex_t** data type is a structure; on other systems, it may be a pointer or another data type.

A mutex must be created once. Calling the **pthread_mutex_init** subroutine more than once with the same *mutex* parameter (for example, in two threads concurrently executing the same code) should be avoided. The second call will fail, returning an **EBUSY** error code. Ensuring the uniqueness of a mutex creation can be done in three ways:

- Calling the **pthread_mutex_init** subroutine prior to the creation of other threads that will use this mutex; in the initial thread, for example.
- Calling the **pthread_mutex_init** subroutine within a one time initialization routine; see One-Time Initializations (“One-Time Initializations” on page 246).
- Using a static mutex initialized by the **PTHREAD_MUTEX_INITIALIZER** static initialization macro; the mutex will have default attributes.

Once the mutex is no longer needed, it should be destroyed by calling the **pthread_mutex_destroy** subroutine. This subroutine may reclaim any storage allocated by the **pthread_mutex_init** subroutine. After having destroyed a mutex, the same **pthread_mutex_t** variable can be reused for creating another mutex. For example, the following code fragment is legal, although not very realistic:

```
pthread_mutex_t mutex;
...
for (i = 0; i < 10; i++) {

        /* creates a mutex */
        pthread_mutex_init(&mutex, NULL);

        /* uses the mutex */

        /* destroys the mutex */
        pthread_mutex_destroy(&mutex);
}
```

Like any system resource that can be shared among threads, a mutex allocated on a thread's stack must be destroyed before the thread is terminated. The threads library maintains a linked list of mutexes; thus if the stack where a mutex is allocated is freed, the list will be corrupted.

Locking and Unlocking Mutexes

A mutex is a simple lock, having two states: locked and unlocked. When it is created, a mutex is unlocked. The **pthread_mutex_lock** subroutine locks the specified mutex:

- If the mutex is unlocked, the subroutine locks it.
- If the mutex is already locked by another thread, the subroutine blocks the calling thread until the mutex is unlocked.
- If the mutex is already locked by the calling thread, the subroutine returns an error.

The **pthread_mutex_trylock** subroutine acts like the **pthread_mutex_lock** subroutine without blocking the calling thread:

- If the mutex is unlocked, the subroutine locks it.
- If the mutex is already locked by any thread, the subroutine returns an error.

The thread that locked a mutex is often called the *owner* of the mutex.

The **pthread_mutex_unlock** subroutine resets the specified mutex to the unlocked state if it is owned by the calling mutex:

- If the mutex was already unlocked, the subroutine returns an error.
- If the mutex was owned by the calling thread, the subroutine unlocks the mutex.
- If the mutex was owned by another thread, the subroutine returns an error.

Because locking does not provide a cancellation point ("Cancellation Points" on page 222), a thread blocked while waiting for a mutex cannot be canceled ("Canceling a Thread" on page 221). Therefore, it is recommended to use mutexes only for short periods of time, like protecting data from concurrent access.

Protecting Data with Mutexes

Mutexes are intended to serve either as a low level primitive from which other thread synchronization functions can be built or as a data protection lock. "Making Complex Synchronization Objects" on page 252 provides more information about implementing long locks and writer-priority readers/writers locks with mutexes.

Mutex Usage Example

Mutexes can be used to protect data from concurrent access. For example, a database application may create several threads to handle several requests concurrently. The database itself is protected by a mutex, called **db_mutex**.

```
/* the initial thread */
pthread_mutex_t mutex;
int i;
...
pthread_mutex_init(&mutex, NULL);    /* creates the mutex      */
for (i = 0; i < num_req; i++)        /* loop to create threads */
        pthread_create(th + i, NULL, rtn, &mutex);
...                                  /* waits end of session   */
pthread_mutex_destroy(&mutex);       /* destroys the mutex     */
...
```

```
/* the request handling thread */
...                                 /* waits for a request  */
pthread_mutex_lock(&db_mutex);      /* locks the database   */
...                                 /* handles the request  */
pthread_mutex_unlock(&db_mutex);    /* unlocks the database */
...
```

The initial thread creates the mutex and all the request handling threads. The mutex is passed to the thread using the parameter of the thread's entry point routine. In a real program, the address of the mutex may be a field of a more complex data structure passed to the created thread.

Avoiding Deadlocks

In AIX, mutexes cannot be re-locked by the same thread. This may not be the case on other systems. To enhance portability of your programs, assume that the following code fragment may produce a deadlock:

```
pthread_mutex_lock(&mutex);
pthread_mutex_lock(&mutex);
```

This kind of deadlock may occur when locking a mutex and then calling a routine that will itself lock the same mutex. For example:

```
pthread_mutex_t mutex;
struct {
        int a;
        int b;
        int c;
} A;

f()
{
        pthread_mutex_lock(&mutex);      /* call 1 */
        A.a++;
        g();
        A.c = 0;
        pthread_mutex_unlock(&mutex);
}

g()
{
        pthread_mutex_lock(&mutex);      /* call 2 */
        A.b += A.a;
        pthread_mutex_unlock(&mutex);    /* call 3 */
}
```

On some non-AIX systems, calling the **f** subroutine would produce a deadlock; call 2 would block the thread, because call 1 already locked the mutex. In AIX, this code fragment would still not have the expected behavior. Call 2 would be unsuccessful, but call 3 would succeed. Thus, when returning for the **g** subroutine, the mutex would already be unlocked and the **A** variable would no longer be protected; when returning from the **f** routine, the **A.c** variable may not contain zero.

To avoid this kind of deadlock or data inconsistency, you should use either one of the following schemes:

- *Fine granularity locking*. Each data atom should be protected by a mutex, locked only by low-level functions. For example, this would result in locking each record of a database. Benefits: high-level functions do not need to care about locking data. Drawbacks: it increases the number of mutexes, and great care should be taken to avoid deadlocks.
- *High-level locking*. Data should be organized into areas, each area protected by a mutex; low-level functions do not need to care about locking. For example, this would result in locking a whole database before accessing it. Benefits: there are few mutexes, and thus few risks of deadlocks. Drawbacks: performance may be bad, especially if many threads want access to the same data.

Deadlocks may also occur when locking mutexes in reverse order. For example, the following code fragment may produce a deadlock between threads A and B:

```
/* Thread A */
pthread_mutex_lock(&mutex1);
pthread_mutex_lock(&mutex2);
/* Thread B */
pthread_mutex_lock(&mutex2);
pthread_mutex_lock(&mutex1);
```

To avoid these kinds of deadlocks, you should ensure that successive mutexes are always locked in the same order.

Using Condition Variables

Condition variables allow threads to wait until some event or condition has occurred. Typically, a program will use three objects:

- A boolean variable, indicating whether the condition is met
- A mutex to serialize the access to the boolean variable
- A condition variable to wait for the condition.

Using a condition variable requires some effort from the programmer. However, condition variables allow the implementation of powerful and efficient synchronization mechanisms. See “Making Complex Synchronization Objects” on page 252 for more information about implementing long locks and semaphores with condition variables.

A condition variable has attributes, which specify the characteristics of the condition. In the current version of AIX, the condition attributes are not used. Therefore, the condition attributes object can be ignored when creating a condition variable.

Condition Attributes Object

Like threads and mutexes, condition variables are created with the help of an attributes object. The condition attributes object is an abstract object, containing at most one attribute, depending on the implementation of POSIX options. It is accessed through a variable of type **pthread_condattr_t**. In AIX, the **pthread_condattr_t** data type is a pointer; on other systems, it may be a structure or another data type.

Condition Attributes Object Creation and Destruction

The mutex attributes object is initialized to default values by the **pthread_condattr_init** subroutine. The attribute is handled by subroutines. The thread attributes object is destroyed by the **pthread_condattr_destroy** subroutine. This subroutine may free storage dynamically allocated by the **pthread_condattr_init** subroutine, depending on the implementation of the threads library.

In the following example, a mutex attributes object is created and initialized with default values, then used and finally destroyed:

```
pthread_condattr_t attributes;
                /* the attributes object is created */
...
if (!pthread_condattr_init(&attributes)) {
                /* the attributes object is initialized */
        ...
                /* using the attributes object */
        ...
        pthread_condattr_destroy(&attributes);
                        /* the attributes object is destroyed */
}
```

The same attributes object can be used to create several condition variables. It can also be modified between two condition variable creations. When the condition variables are created, the attributes object can be destroyed without affecting the condition variables created with it.

Condition Attribute

In AIX, no condition attribute is defined. Condition attributes depend on POSIX options that are not implemented in AIX. However, the following attribute may be defined on other systems:

Process-shared Specifies the process sharing of a condition variable. This attribute depends on the process sharing POSIX option.

See “Advanced Attributes” on page 250 for more information about the process-shared attribute.

Creating and Destroying Condition Variables

A condition variable is created by calling the **pthread_cond_init** subroutine. You may specify a condition attributes object. If you specify a **NULL** pointer, the condition variable will have the default attributes. Thus, the code fragment:

```
pthread_cond_t cond;
pthread_condattr_t attr;
...
pthread_condattr_init(&attr);
pthread_cond_init(&cond, &attr);
pthread_condattr_destroy(&attr);
```

is equivalent to:

```
pthread_cond_t cond;
...
pthread_cond_init(&cond, NULL);
```

The ID of the created condition variable is returned to the calling thread through the *condition* parameter. The condition ID is an opaque object; its type is **pthread_cond_t**. In AIX, the **pthread_cond_t** data type is a structure; on other systems it may be a pointer or another data type.

A condition variable must be created once. Calling the **pthread_cond_init** subroutine more than once with the same *condition* parameter (for example, in two threads concurrently executing the same code) should be avoided. The second call will fail, returning an **EBUSY** error code. Ensuring the uniqueness of a condition variable creation can be done in three ways:

- Calling the **pthread_cond_init** subroutine prior to the creation of other threads that will use this variable; in the initial thread, for example.
- Calling the **pthread_cond_init** subroutine within a one-time initialization routine (“One-Time Initializations” on page 246).
- Using a static condition variable initialized by the **PTHREAD_COND_INITIALIZER** static initialization macro; the condition variable will have default attributes.

Once the condition variable is no longer needed, it should be destroyed by calling the **pthread_cond_destroy** subroutine. This subroutine may reclaim any storage allocated by the **pthread_cond_init** subroutine. After having destroyed a condition variable, the same **pthread_cond_t** variable can be reused for creating another condition. For example, the following code fragment is legal, although not very realistic:

```
pthread_cond_t cond;
...
for (i = 0; i < 10; i++) {

        /* creates a condition variable */
        pthread_cond_init(&cond, NULL);
```

```
        /* uses the condition variable */

        /* destroys the condition */
        pthread_cond_destroy(&cond);
}
```

Like any system resource that can be shared among threads, a condition variable allocated on a thread's stack must be destroyed before the thread is terminated. The threads library maintains a linked list of condition variables; thus if the stack where a mutex is allocated is freed, the list will be corrupted.

Using Condition Variables

A condition variable must always be used together with a mutex. The same mutex must be used for the same condition variable, even for different threads. It is possible to bundle in a structure the condition, the mutex, and the condition variable, as shown in the following code fragment:

```
struct condition_bundle_t {
        int             condition_predicate;
        pthread_mutex_t  condition_lock;
        pthread_cond_t   condition_variable;
};
```

See "Synchronizing Threads with Condition Variables" on page 235 for more information about using the condition predicate.

Waiting for a Condition

The mutex protecting the condition must be locked before waiting for the condition. A thread can wait for a condition to be signaled by calling the **pthread_cond_wait** or **pthread_cond_timedwait** subroutine. The subroutine atomically unlocks the mutex and blocks the calling thread until the condition is signaled. When the call returns, the mutex is locked again.

The **pthread_cond_wait** subroutine blocks the thread indefinitely. If the condition is never signaled, the thread never wakes up. Because the **pthread_cond_wait** subroutine provides a cancellation point, the only way to get out of this deadlock is to cancel the blocked thread, if cancelability is enabled. For more information, see "Canceling a Thread" on page 221.

The **pthread_cond_timedwait** subroutine blocks the thread only for a given period of time. This subroutine has an extra parameter, *timeout*, specifying an absolute date where the sleep must end. The *timeout* parameter is a pointer to a **timespec** structure. This data type is also called **timestruc_t**. It contains two fields:

`tv_sec`	A long unsigned integer, specifying seconds
`tv_nsec`	A long integer, specifying nanoseconds.

Typically, the **pthread_cond_timedwait** subroutine is used in the following manner:

```
struct timespec timeout;
...
time(&timeout.tv_sec);
timeout.tv_sec += MAXIMUM_SLEEP_DURATION;
pthread_cond_timedwait(&cond, &mutex, &timeout);
```

The *timeout* parameter specifies an absolute date. The previous code fragment shows how to specify a duration rather than an absolute date.

To use **pthread_cond_timedwait** with an absolute date, you can use the **mktime** subroutine to calculate the value of the `tv_sec` field of the **timespec** structure. In the following example, the thread will wait for the condition until 08:00 January 1, 2001, local time:

```
struct tm       date;
time_t          seconds;
struct timespec timeout;
...
date.tm_sec = 0;
date.tm_min = 0;
date.tm_hour = 8;
date.tm_mday = 1;
date.tm_mon = 0;         /* the range is 0-11 */
date.tm_year = 101;      /* 0 is 1900 */
date.tm_wday = 1;        /* this field can be omitted -
                            but it will really be a Monday! */
date.tm_yday = 0;        /* first day of the year */
date.tm_isdst = daylight;
        /* daylight is an external variable - we are assuming
           that daylight savings time will still be used... */
seconds = mktime(&date);
timeout.tv_sec = (unsigned long)seconds;
timeout.tv_nsec = 0L;
pthread_cond_timedwait(&cond, &mutex, &timeout);
```

The **pthread_cond_timedwait** subroutine also provides a cancellation point, although the sleep is not indefinite. Thus, a sleeping thread can be canceled, whether the sleep has a timeout or not.

Signaling a Condition

A condition can be signaled by calling either the **pthread_cond_signal** or the **pthread_cond_broadcast** subroutine.

The **pthread_cond_signal** subroutine wakes up at least one thread that is currently blocked on the specified condition. The awoken thread is chosen according to the scheduling policy; it is the thread with the most-favored scheduling priority (see “Scheduling Policy and Priority” on page 240) . It may happen on multiprocessor systems, or some non-AIX systems, that more than one thread is woken up. Do not assume that this subroutine wakes up exactly one thread.

The **pthread_cond_broadcast** subroutine wakes up every thread that is currently blocked on the specified condition. However, a thread can start waiting on the same condition just after the call to the subroutine returns.

A call to these routines always succeeds, unless an invalid *cond* parameter is specified. This does not mean that a thread has been awakened. Furthermore, signaling a condition is not remembered by the library. For example, consider a condition C. No thread is waiting on this condition. At time t, thread 1 signals the condition C. The call is successful although no thread is woken up. At time t+1, thread 2 calls the **pthread_cond_wait** subroutine with C as *cond* parameter. Thread 2 is blocked. If no other thread signals C, thread 2 may wait until the process terminates.

A way to avoid this kind of deadlock is to check the **EBUSY** error code returned by the **pthread_cond_destroy** subroutine when destroying the condition variable, as in the following code fragment:

```
while (pthread_cond_destroy(&cond) == EBUSY) {
        pthread_cond_broadcast(&cond);
        pthread_yield();
}
```

The **pthread_yield** subroutine gives the opportunity to another thread to be scheduled, one of the awoken threads for example. See “Threads Scheduling” on page 240 for more information about the **pthread_yield** subroutine.

The **pthread_cond_wait** and the **pthread_cond_broadcast** subroutines must not be used within a signal handler. To provide a convenient way for a thread to await a signal, the threads library provides the **sigwait** subroutine. See “Signal Management” on page 256 for more information about the **sigwait** subroutine.

Synchronizing Threads with Condition Variables

Condition variables are used to wait until a particular predicate becomes true. This predicate is set by another thread, usually the one that signals the condition.

Condition Wait Semantics

A predicate must be protected by a mutex. When waiting for a condition, the wait subroutine (either **pthread_cond_wait** or **pthread_cond_timedwait**) atomically unlocks the mutex and blocks the thread. When the condition is signaled, the mutex is relocked and the wait subroutine returns. It is important to note that when the subroutine returns without error, the predicate may still be false.

The reason is that more than one thread may be awoken: either a thread called the **pthread_cond_broadcast** subroutine, or an unavoidable race between two processors simultaneously woke two threads. The first thread locking the mutex will block all other awoken threads in the wait subroutine until the mutex is unlocked by the program. Thus, the predicate may have changed when the second thread gets the mutex and returns from the wait subroutine.

In general, whenever a condition wait returns, the thread should re-evaluate the predicate to determine whether it can safely proceed, should wait again, or should declare a timeout. A return from the wait subroutine does not imply that the predicate is either true or false.

It is recommended that a condition wait be enclosed in a ″while loop″ that checks the predicate. The following code fragment provides a basic implementation of a condition wait.

```
pthread_mutex_lock(&condition_lock);
while (condition_predicate == 0)
        pthread_cond_wait(&condition_variable, &condition_lock);
...
pthread_mutex_unlock(&condition_lock);
```

Timed Wait Semantics

When the **pthread_cond_timedwait** subroutine returns with the timeout error, the predicate may be true. This is due to another unavoidable race between the expiration of the timeout and the predicate state change.

Just as for non-timed wait, the thread should re-evaluate the predicate when a timeout occurred to determine whether it should declare a timeout or should proceed anyway. It is recommended to carefully check all possible cases when the **pthread_cond_timedwait** subroutine returns. The following code fragment shows how such checking could be implemented in a robust program:

```
int result = CONTINUE_LOOP;

pthread_mutex_lock(&condition_lock);
while (result == CONTINUE_LOOP) {
        switch (pthread_cond_timedwait(&condition_variable,
                &condition_lock, &timeout)) {
                case 0:
                if (condition_predicate)
                        result = PROCEED;
                break;
                case ETIMEDOUT:
                result = condition_predicate ? PROCEED : TIMEOUT;
                break;
```

```
                default:
                result = ERROR;
                break;
        }
}
...
pthread_mutex_unlock(&condition_lock);
```

The **result** variable can be used to choose an action. The statements preceding the unlocking of the mutex should be as quick as possible, because a mutex should not be held for long periods of time.

Specifying an absolute date in the *timeout* parameter allows easy implementation of real-time behavior. An absolute timeout does not need to be recomputed if it is used multiple times in a loop, such as that enclosing a condition wait. For cases where the system clock is advanced discontinuously by an operator, using an absolute timeout ensures that the timed wait will end as soon as the system time specifies a date later than the *timeout* parameter.

Condition Variables Usage Example

The following example provides the source code for a synchronization point routine. A synchronization point is a given point in a program where different threads must wait until all threads (or at least a certain number of threads) have reached that point.

A synchronization point can simply be implemented by a counter, which is protected by a lock, and a condition variable. Each thread takes the lock, increments the counter, and waits for the condition to be signaled if the counter did not reach its maximum. Otherwise, the condition is broadcast, and all threads can proceed. The last thread calling the routine broadcasts the condition.

```
#define SYNC_MAX_COUNT  10

void SynchronizationPoint()
{
        /* use static variables to ensure initialization */
        static mutex_t sync_lock = PTHREAD_MUTEX_INITIALIZER;
        static cond_t  sync_cond = PTHREAD_COND_INITIALIZER;
        static int sync_count = 0;

        /* lock the access to the count */
        pthread_mutex_lock(&sync_lock);

        /* increment the counter */
        sync_count++;

        /* check if we should wait or not */
        if (sync_count < SYNC_MAX_COUNT)

                /* wait for the others */
                pthread_cond_wait(&sync_cond, &sync_lock);

        else

                /* broadcast that everybody reached the point */
                pthread_cond_broadcast(&sync_cond);

        /* unlocks the mutex - otherwise only one thread
                will be able to return from the routine! */
        pthread_mutex_unlock(&sync_lock);
}
```

This routine has some limitations: it can be used only once, and the number of threads that will call the routine is coded by a symbolic constant. However, this example shows a basic usage of condition variables. More complex usage can be found in “Making Complex Synchronization Objects” on page 252.

Joining Threads

Joining a thread means waiting for it to terminate. It can be seen as a specific usage of condition variables.

Waiting for a Thread

The **pthread_join** subroutine provides a simple mechanism allowing a thread to wait for another thread to terminate. More complex conditions, such as waiting for multiple threads to terminate, can be implemented by the programmer using condition variables. See “Synchronizing Threads with Condition Variables” on page 235 for more information.

Calling the pthread_join Subroutine

The **pthread_join** subroutine blocks the calling thread until the specified thread terminates. The target thread (the thread whose termination is awaited) must not be detached. If the target thread is already terminated, but not detached, the **pthread_join** subroutine returns immediately. Once a target thread has been joined, it is automatically detached, and its storage can be reclaimed.

The following table indicates the two possible cases when a thread calls the **pthread_join** subroutine, depending on the state and the detachstate attribute of the target thread.

	Undetached target	**Detached target**
Target is still running	The caller is blocked until the target is terminated.	The call returns immediately indicating an error.
Target is terminated	The call returns immediately indicating a successful completion.	

A thread cannot join itself - a deadlock would occur and it is detected by the library. However, two threads may try to join each other; they will deadlock. This situation is not detected by the library.

Multiple Joins

It is possible for several threads to join the same target thread, if the target is not detached. The success of this operation depends on the order of the calls to the **pthread_join** subroutine and the moment when the target thread terminates.

- Any call to the **pthread_join** subroutine occurring before the target thread’s termination blocks the calling thread.
- When the target thread terminates, all blocked threads are awoken, and the target thread is automatically detached.
- Any call to the **pthread_join** subroutine occurring after the target thread’s termination will fail, because the thread is detached by the previous join.
- If no thread called the **pthread_join** subroutine before the target thread’s termination, the first call to the **pthread_join** subroutine will return immediately, indicating a successful completion, and any further call will fail.

Join Example

The following example is an enhanced version of the first multi-threaded program. The program ends after exactly five messages in each language are displayed. This is done by blocking the initial thread until the ″writer″ threads exit.

```
#include <pthread.h>    /* include file for pthreads - the 1st */
#include <stdio.h>      /* include file for printf()           */

void *Thread(void *string)
{
        int i;

        /* writes five messages and exits */
        for (i=0; i<5; i++)
                printf("%s\n", (char *)string);
        pthread_exit(NULL);
}
```

```
int main()
{
        char *e_str = "Hello!";
        char *f_str = "Bonjour !";

        pthread_attr_t attr;
        pthread_t e_th;
        pthread_t f_th;

        int rc;
        /* creates the right attribute */
        pthread_attr_init(&attr);
        pthread_attr_setdetachstate(&attr,
                PTHREAD_CREATE_UNDETACHED);
        /* creates both threads */
        rc = pthread_create(&e_th, &attr, Thread, (void *)e_str);
        if (rc)
                exit(-1);
        rc = pthread_create(&f_th, &attr, Thread, (void *)f_str);
        if (rc)
                exit(-1);
        pthread_attr_destroy(&attr);
        /* joins the threads */
        pthread_join(e_th, NULL);
        pthread_join(f_th, NULL);

        pthread_exit(NULL);
}
```

Returning Information from a Thread

The **pthread_join** subroutine also allows a thread to return information to another thread. When a thread calls the **pthread_exit** subroutine or when it returns from its entry-point routine, it returns a pointer (see “Exiting a Thread” on page 220). This pointer is stored as long as the thread is not detached, and the **pthread_join** subroutine can return it.

For example, a multi-threaded **grep** command may choose the following implementation. The initial thread creates one thread per file to scan, each thread having the same entry point routine. It then waits for all threads to be terminated. Each ″scanning″ thread stores the found lines in a dynamically allocated buffer and returns a pointer to this buffer. The initial thread prints out each buffer and frees it.

```
/* "scanning" thread */
...
buffer = malloc(...);
        /* finds the search pattern in the file
           and stores the lines in the buffer   */
return (buffer);
/* initial thread */
...
for (/* each created thread */) {
        void *buf;
        pthread_join(thread, &buf);
        if (buf != NULL) {
                /* print all the lines in the buffer,
                        preceded by the filename of the thread */
                free(buf);
        }
}
...
```

If the target thread is canceled, the **pthread_join** subroutine returns a value of -1 cast into a pointer (see “Canceling a Thread” on page 221). Because -1 cannot be a pointer value, getting -1 as returned pointer from a thread means that the thread was canceled.

The returned pointer can point to any kind of data. Care must be taken concerning the storage class of the data the pointer refers to. The pointer must be still valid after the thread was terminated and its storage reclaimed. Therefore, returning a “Thread-Specific Data” on page 247 value should be avoided, because the destructor routine is called when the thread’s storage is reclaimed.

Returning a pointer to dynamically allocated storage to several threads should also be handled with care. Consider the following code fragment:

```
void *returned_data;
...
pthread_join(target_thread, &returned_data);
/* retrieves information from returned_data */
free(returned_data);
```

When executed by only one thread, the **returned_data** pointer is freed as it should be. If several threads execute this code fragment concurrently, the **returned_data** pointer is freed several times; this must be avoided. A solution may consist in using a flag, protected by a mutex, to signal that the **returned_data** pointer was freed. The line:

```
free(returned_data);
```

would thus be replaced by the lines (assuming the **flag** variable is initially 0)

```
/* lock - entering a critical region, no other thread should
                run this portion of code concurrently */
if (!flag) {
        free(returned_data);
        flag = 1;
}
/* unlock - exiting the critical region */
```

where a mutex (“Using Mutexes” on page 227) can be used for locking the access to the critical region. This ensures that the **returned_data** pointer is freed only once.

When returning a pointer to dynamically allocated storage to several threads all executing different code, you must ensure that exactly one thread frees the pointer.

List of Synchronization Subroutines

pthread_mutex_destroy	Deletes a mutex.
pthread_mutex_init	Initializes a mutex and sets its attributes.
PTHREAD_MUTEX_INITIALIZER	Initializes a static mutex with default attributes.
pthread_mutex_lock or **pthread_mutex_trylock**	Locks a mutex.
pthread_mutex_unlock	Unlocks a mutex.
pthread_mutexattr_destroy	Deletes a mutex attributes object.
pthread_mutexattr_init	Creates a mutex attributes object and initializes it with default values.
pthread_cond_destroy	Deletes a condition variable.
pthread_cond_init	Initializes a condition variable and sets its attributes.
PTHREAD_COND_INITIALIZER	Initializes a static condition variable with default attributes.
pthread_cond_signal or **pthread_cond_broadcast**	Unblocks one or more threads blocked on a condition.
pthread_cond_wait or **pthread_cond_timedwait**	Blocks the calling thread on a condition.
pthread_condattr_destroy	Deletes a condition attributes object.
pthread_condattr_init	Creates a condition attributes object and initializes it with default values.

Scheduling Overview

Threads are the schedulable entity. The threads library provides several facilities to handle and control the scheduling of threads. It also provides facilities to control the scheduling of threads during synchronization operations such as locking a mutex.

The following information will help you in using the scheduling facilities:

- "Threads Scheduling"
- "Synchronization Scheduling" on page 243
- "List of Scheduling Subroutines" on page 245

Threads Scheduling

Each thread has its own set of scheduling parameters. These parameters can be set using the thread attributes object before the thread's creation. They can also be dynamically set during the thread's execution.

Basic Scheduling Facilities

Controlling the scheduling of a thread is often a complicated task. Because the scheduler handles all threads systemwide, the scheduling parameters of a thread interact with those of all other threads in the process and in the other processes. The following facilities are the first to be used if you want to control the scheduling of a thread.

Inheritsched Attribute

The inheritsched attribute of the thread attributes object specifies how the thread's scheduling attributes will be defined. It may have one of the following values:

PTHREAD_INHERIT_SCHED	Specifies that the new thread will get the scheduling attributes (schedpolicy and schedparam attributes) of its creating thread. Scheduling attributes defined in the attributes object are ignored.
PTHREAD_EXPLICIT_SCHED	Specifies that the new thread will get the scheduling attributes defined in this attributes object.

The default value of the inheritsched attribute is **PTHREAD_INHERIT_SCHED**. The attribute is set by calling the **pthread_attr_setinheritsched** subroutine. The current value of the attribute is returned by calling the **pthread_attr_getinheritsched** subroutine.

To set the scheduling attributes of a thread in the thread attributes object, the inheritsched must first be set to **PTHREAD_EXPLICIT_SCHED**. Otherwise, the attributes object scheduling attributes are ignored.

Scheduling Policy and Priority

The threads library provides three scheduling policies:

SCHED_FIFO	First-in first-out (FIFO) scheduling. Each thread has a fixed priority; when multiple threads have the same priority level, they run to completion in FIFO order.
SCHED_RR	Round-robin (RR) scheduling. Each thread has a fixed priority; when multiple threads have the same priority level, they run for a fixed time slice in FIFO order.
SCHED_OTHER	Default AIX scheduling. Each thread has a initial priority that is dynamically modified by the scheduler, according to the thread's activity; thread execution is time-sliced. On other systems, this scheduling policy may be different.

The default scheduling policy for threads is **SCHED_OTHER**.

The priority is an integer value, in the range from 1 to 127. 1 is the least-favored priority, 127 is the most-favored. Priority level 0 cannot be used: it is reserved for the system. Note that in AIX, the kernel inverts the priority levels. For the AIX kernel, the priority is in the range from 0 to 127, where 0 is the most favored priority and 127 the least-favored. Commands, such as the **ps** command, report the kernel priority.

The threads library handles the priority through a **sched_param** structure, defined in the **sys/sched.h** header file. Currently, this structure contains two fields:

`sched_priority`	Specifies the priority.
`sched_policy`	This field is ignored by the threads library and should not be used.

In the future, other fields may be defined for other scheduling characteristics.

Setting the Scheduling Policy and Priority at Creation Time

The scheduling policy can be set when creating a thread by setting the schedpolicy attribute of the thread attributes object. The **pthread_attr_setschedpolicy** subroutine sets the scheduling policy to one of the three previously defined scheduling policies. The current value of the schedpolicy attribute of a thread attributes object can be obtained by the **pthread_attr_getschedpolicy** subroutine.

The scheduling priority can be set at creation time of a thread by setting the schedparam attribute of the thread attributes object. The **pthread_attr_setschedparam** subroutine sets the value of the schedparam attribute, copying the value of the specified structure. The **pthread_attr_getschedparam** subroutine gets the schedparam attribute.

In the following code fragment, a thread is created with the round-robin scheduling policy, using a priority level of 3:

```
sched_param schedparam;

schedparam.sched_priority = 3;
pthread_attr_init(&attr);
pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
pthread_attr_setschedpolicy(&attr, SCHED_RR);
pthread_attr_setschedparam(&attr, &schedparam);
pthread_create(&thread, &attr, &start_routine, &args);
pthread_attr_destroy(&attr);
```

See “Inheritsched Attribute” on page 240 to get more information about the inheritsched attribute.

Setting the Scheduling Attributes at Execution Time

The current schedpolicy and schedparam attributes of a thread are returned by the **pthread_getschedparam** subroutine. These attributes can be set by calling the **pthread_setschedparam** subroutine. If the target thread is currently running on a processor, the new scheduling policy and priority will be implemented the next time the thread is scheduled. If the target thread is not running, it may be scheduled immediately at the end of the subroutine call.

For example, consider a thread T that is currently running with RR policy at the moment the schedpolicy attribute of T is changed to FIFO. T will run until the end of its time slice, at which time its scheduling attributes are then re-evaluated. If no threads have higher priority, T will be rescheduled, even before other threads having the same priority. Consider a second example where a low-priority thread is not running. If this thread’s priority is raised by another thread calling **pthread_setschedparam**, the target thread will be scheduled immediately if it is the highest priority runnable thread.

Note: Both subroutines use two parameters: a *policy* parameter and a **sched_param** structure. Although this structure contains a `sched_policy` field, programs should not use it. The subroutines use the *policy* parameter to pass the scheduling policy and ignore the `sched_policy` field.

Considerations about Scheduling Policies

Applications should use the default scheduling policy, unless a specific application requires the use of a fixed-priority scheduling policy.

Using the RR policy ensures that all threads having the same priority level will be scheduled equally, regardless of their activity. This can be useful in programs where threads have to read sensors or write actuators.

Using the FIFO policy should be done with great care. A thread running with FIFO policy runs to completion, unless it is blocked by some calls, such as performing input and output operations. A high-priority FIFO thread may not be preempted and can affect the global performance of the system. For example, threads doing intensive calculations, such as inverting a large matrix, should never run with FIFO policy.

The setting of scheduling policy and priority is also influenced by the contention scope of threads. Using the FIFO or the RR policy may not always be allowed. See “Impacts of Contention Scope on Scheduling” on page 243 for more information.

Contention Scope

The threads library defines two possible contention scopes:

PTHREAD_SCOPE_PROCESS	Process (or local) contention scope. Specifies that the thread will be scheduled against all other local contention scope threads in the process.
PTHREAD_SCOPE_SYSTEM	System (or global) contention scope. Specifies that the thread will be scheduled against all other threads in the system.

See “Threads Implementation” on page 162 for more information about contention scope.

Setting the Contention Scope

The contention scope can only be set before thread creation by setting the contention-scope attribute of a thread attributes object. The **pthread_attr_setscope** subroutine sets the value of the attribute; the **pthread_attr_getscope** returns it.

The contention scope is only meaningful in a mixed-scope M:N library implementation. A single-scope 1:1 library implementation, as in Pre-AIX 4.3, always returns an error when trying to set the contention-scope attribute to **PTHREAD_SCOPE_PROCESS**, because all threads have system contention scope. This is the easiest way to test the implementation of a threads library. A **TestImplementation** routine could be written as follows:

```
int TestImplementation()
{
        pthread_attr_t a;
        int result;

        pthread_attr_init(&a);
        switch (pthread_attr_setscope(&a, PTHREAD_SCOPE_PROCESS))
        {
                case 0:          result = LIB_MN; break;
                case ENOTSUP:    result = LIB_11; break;
                case ENOSYS:     result = NO_PRIO_OPTION; break;
                default:         result = ERROR; break;
        }
```

```
        pthread_attr_destroy(&a);
        return result;
}
```

Prior to AIX 4.3, this routine would return **LIB_11**.

In AIX 4.3, this routine returns **LIB_MN**.

Impacts of Contention Scope on Scheduling

The contention scope of a thread influences its scheduling. Each system contention scope thread is bound to one kernel thread. Thus changing the scheduling policy and priority of a global user thread results in changing the scheduling policy and priority of the underlying kernel thread.

In AIX, only kernel threads with root authority can use a fixed-priority scheduling policy (FIFO or RR). The following code:

```
schedparam.sched_priority = 3;
pthread_setschedparam(pthread_self(), SCHED_FIFO, schedparam);
```

will always return the **EPERM** error code if the calling thread has system contention scope but does not have root authority. It would not fail, if the calling thread had process contention scope. One does not need to have root authority to control the scheduling parameters of user threads with process contention scope.

Local user thread can set any scheduling policy and priority, within the valid range of values. However, two threads having the same scheduling policy and priority but having different contention scope will not be scheduled in the same way. Threads having process contention scope are executed by kernel threads whose scheduling parameters are set by the library.

sched_yield Subroutine

The **sched_yield** subroutine is the equivalent for threads of the **yield** subroutine. It forces the calling thread to relinquish the use of its processor. It gives other threads a chance to be scheduled. The next scheduled thread may belong to the same process as the calling thread or to another process. The **yield** subroutine must not be used in a multi-threaded program.

The interface pthread_yield subroutine is not available in XOPEN VERSION 5.

Synchronization Scheduling

Programmers may want to control the execution scheduling of threads when there are constraints, especially time constraints, that require certain threads to be executed faster than other ones. Synchronization objects, such as mutexes, may block even high-priority threads. In some cases, undesirable behavior, known as *priority inversion*, may occur. The threads library provides a facility, the *mutex protocols*, to avoid priority inversions.

Priority Inversion

Priority inversion occurs when a low-priority thread holds a mutex, blocking a high-priority thread. Due to its low priority, the mutex owner may hold the mutex for an unbounded duration. As a result, it becomes impossible to guarantee thread deadlines.

The following example illustrates a typical priority inversion. To make the example easier to understand, only the case of a uniprocessor system is considered. Priority inversions also occur on multiprocessor systems in a similar way.

A mutex M is used to protect some common data. Thread A has a priority level of 100. It should be scheduled very often. Thread B has a priority level of 20. It is a background thread. Other threads in the process have priority levels around 60. A code fragment from thread A is:

```
pthread_mutex_lock(&M);             /* 1 */
...
pthread_mutex_unlock(&M);
```

A code fragment from thread B is:

```
pthread_mutex_lock(&M);          /* 2 */
...
fprintf(...);                    /* 3 */
...
pthread_mutex_unlock(&M);
```

Consider the following execution chronology. Thread B is scheduled and executes line 2. When executing line 3, thread B is preempted by thread A. Thread A executes line 1 and is blocked, because the mutex M is held by thread B. Thus, other threads in the process are scheduled. Because thread B has a very low priority, it may not be rescheduled for a long period, blocking thread A although thread A has a very high priority.

Mutex Protocols

To avoid priority inversions, two mutex protocols are provided by the threads library:

- “Priority Inheritance Protocol”, sometimes called basic priority inheritance protocol
- “Priority Protection Protocol”l, sometimes called priority ceiling protocol emulation.

Both protocols increase the priority of a thread holding a specific mutex, so that deadlines can be guaranteed. Furthermore, when correctly used, mutex protocols can prevent mutual deadlocks. Mutex protocols are individually assigned to mutexes.

Priority Inheritance Protocol

In the priority inheritance protocol, the mutex holder inherits the priority of the highest priority blocked thread. When a thread tries to lock a mutex using this protocol and is blocked, the mutex owner temporarily receives the blocked thread’s priority, if that priority is higher than the owner’s. It recovers its original priority when it unlocks the mutex.

Priority Protection Protocol

In the priority protection protocol, each mutex has a *priority ceiling*. It is a priority level within the valid range of priorities. When a thread owns a mutex, it temporarily receives the mutex priority ceiling, if the ceiling is higher than its own priority. It recovers its original priority when it unlocks the mutex. The priority ceiling should have the value of the highest priority of all threads that may lock the mutex. Otherwise, priority inversions or even deadlocks may occur, and the protocol would be inefficient.

Choosing a Mutex Protocol

The choice of a mutex protocol is made by setting attributes when creating a mutex. See “Protocol Attribute” for more information. “Inheritance or Protection” on page 245 provides guidelines for choosing a protocol.

Protocol Attribute

The mutex protocol is controlled through an attribute: the protocol attribute. This attribute can be set in the mutex attributes object using the **pthread_mutexattr_getprotocol** and **pthread_mutexattr_setprotocol** subroutines. The protocol attribute can have one of the following values:

PTHREAD_PRIO_NONE	Denotes no protocol. This is the default value.
PTHREAD_PRIO_INHERIT	Denotes the priority inheritance protocol.
PTHREAD_PRIO_PROTECT	Denotes the priority protection protocol.

The priority protection protocol uses one additional attribute: the prioceiling attribute. This attribute contains the priority ceiling of the mutex. The prioceiling attribute can be controlled in the mutex attributes object, using the **pthread_mutexattr_getprioceiling** and **pthread_mutexattr_setprioceiling** subroutines.

The prioceiling attribute of a mutex can also be dynamically controlled using the **pthread_mutex_getprioceiling** and **pthread_mutex_setprioceiling** subroutines. Note that when dynamically changing the priority ceiling of a mutex, the mutex is locked by the library; it should not be held by the thread calling the **pthread_mutex_setprioceiling** subroutine to avoid a deadlock. Dynamically setting the priority ceiling of a mutex can be useful when increasing the priority of a thread.

The implementation of mutex protocols is optional. Each protocol is a POSIX option. See "Threads Library Options" on page 261 for more information about the priority inheritance and the priority protection POSIX options.

Inheritance or Protection

Both protocols are similar and result in promoting the priority of the thread holding the mutex. If both protocols are available, a choice must be made. This information will help the programmer in choosing a protocol.

The choice depends on whether the priorities of the threads that will lock the mutex are available to the programmer creating the mutex. Typically, mutexes defined by a library and used by application threads will use the inheritance protocol, whereas mutexes created within the application program will use the protection protocol.

In performance-critical programs, performance considerations may also influence the choice. In most implementations, especially in AIX, changing the priority of a thread results in making a system call. Therefore, the two mutex protocols differ in the amount of system calls they generate.

- Using the inheritance protocol, a system call is made each time a thread is blocked when trying to lock the mutex.
- Using the protection protocol, one system call is always made each time the mutex is locked by a thread.

In most performance-critical programs, the inheritance protocol should be chosen, because mutexes are low contention objects. Mutexes are not held for long periods of time; thus, it is not likely that threads are blocked when trying to lock them.

List of Scheduling Subroutines

pthread_attr_getschedparam	Returns the value of the schedparam attribute of a thread attributes object.
pthread_attr_setschedparam	Sets the value of the schedparam attribute of a thread attributes object.
pthread_getschedparam	Returns the value of the schedpolicy and schedparam attributes of a thread.
pthread_yield	Forces the calling thread to relinquish use of its processor.

Threads Advanced Features

The threads library provides some advanced features to be used by trained programmers. These features are helpful to perform special tasks.

One-Time Initializations

Some C libraries are designed for dynamic initialization. That is, the global initialization for the library is performed when the first procedure in the library is called. In a single-threaded program, this is usually implemented using a static variable whose value is checked on entry to each routine, as in the following code fragment:

```
static int isInitialized = 0;
extern void Initialize();

int function()
{
        if (isInitialized == 0) {
                Initialize();
                isInitialized = 1;
        }
        ...
}
```

For dynamic library initialization in a multi-threaded program a simple initialization flag is not sufficient; this flag must be protected against modification by multiple threads simultaneously calling a library function. Protecting the flag requires the use of a mutex; however, mutexes must be initialized before they are used. Ensuring that the mutex is only initialized once requires a recursive solution to this problem.

To keep the same structure in a multi-threaded program a new subroutine, **pthread_once**, is provided by the threads library. Otherwise, library initialization must be accomplished by an explicit call to a library exported initialization function prior to any use of the library. This subroutine also provides an alternative for initializing mutexes and condition variables.

One-Time Initialization Object

The uniqueness of the initialization is ensured by an object, the one-time initialization object, or once block. It is a variable having the **pthread_once_t** data type. In AIX and most other implementations of the threads library, the **pthread_once_t** data type is a structure.

A one-time initialization object is typically a global variable. It must be initialized with the **PTHREAD_ONCE_INIT** macro, as in the following example:

```
static pthread_once_t once_block = PTHREAD_ONCE_INIT;
```

The initialization can also be done in the initial thread or in any other thread. Several one time initialization objects can be used in the same program. The only requirement is that the one-time initialization object be initialized with the macro.

One-Time Initialization Routine

The **pthread_once** subroutine calls the specified initialization routine associated with the specified one-time initialization object if it is the first time it is called; otherwise, it does nothing. The same initialization routine must always be used with the same one-time initialization object. The initialization routine must have the following prototype:

```
void init_routine();
```

The **pthread_once** subroutine does not provide a cancellation point. However, the initialization routine may provide cancellation points, and, if cancelability is enabled, the first thread calling the **pthread_once** subroutine may be canceled during the execution of the initialization routine. In this case, the routine is not considered as executed, and the next call to the **pthread_once** subroutine would result in recalling the initialization routine.

It is recommended to use cleanup handlers in one-time initialization routines, especially when performing non-idempotent operations, such as opening a file, locking a mutex, or allocating memory. For more information, see “Using Cleanup Handlers” on page 225.

One-time initialization routines can be used for initializing mutexes or condition variables or to perform dynamic initialization. The code fragment shown above would be written in a multi-threaded library as follows:

```
static pthread_once_t once_block = PTHREAD_ONCE_INIT;
extern void Initialize();

int function()
{
        pthread_once(&once_block, Initialize);
        ...
}
```

Thread-Specific Data

Many applications require that certain data be maintained on a per-thread basis across function calls. For example, a multi-threaded **grep** command using one thread for each file needs to have thread-specific file handlers and list of found strings. The thread-specific data interface is provided by the threads library to meet these needs.

Thread-specific data may be viewed as a two-dimensional array of values, with keys serving as the row index and thread IDs as the column index. A thread-specific data *key* is an opaque object, of type **pthread_key_t**. The same key can be used by all threads in a process. Although all threads use the same key, they set and access different thread-specific data values associated with that key. Thread-specific data are void pointers. This allows referencing any kind of data, such as dynamically allocated strings or structures.

In the following figure, thread T2 has a thread-specific data value of 12 associated with the key K3. Another thread T4 has the value 2 associated with the same key.

Table 1. Thread-Specific Data Array

		Threads			
		T1	**T2**	**T3**	**T4**
Keys	**K1**	6	56	4	1
	K2	87	21	0	9
	K3	23	**12**	61	2
	K4	11	76	47	88

Creating and Destroying Keys

Thread-specific data keys must be created before being used. Their values can be automatically destroyed when the corresponding threads terminate. A key can also be destroyed upon request to reclaim its storage.

Key Creation

A thread-specific data key is created by calling the **pthread_key_create** subroutine. This subroutine returns a key. The thread-specific data is set to a value of **NULL** for all threads, including threads not yet created.

For example, consider two threads A and B. Thread A performs the following operations in chronological order:

1. Create a thread-specific data key K.

   Threads A and B can use the key K. The value for both threads is **NULL**.
2. Create a thread C.

   Thread C can also use the key K. The value for thread C is **NULL.**

The number of thread-specific data keys is limited to 508 per process. This number can be retrieved by the **PTHREAD_KEYS_MAX** symbolic constant.

The **pthread_key_create** subroutine must be called only once. Otherwise, two different keys are created. For example, consider the following code fragment:

```
/* a global variable */
static pthread_key_t theKey;

/* thread A */
...
pthread_key_create(&theKey, NULL);   /* call 1 */
...

/* thread B */
...
pthread_key_create(&theKey, NULL);   /* call 2 */
...
```

Threads A and B run concurrently, but call 1 happens before call 2. Call 1 will create a key K1 and store it in the **theKey** variable. Call 2 will create another key K2, and store it also in the **theKey** variable, thus overriding K1. As a result, thread A will use K2, assuming it is K1. This situation should be avoided for two reasons:

- Key K1 is lost, thus its storage will never be reclaimed until the process terminates. Because the number of keys is limited, you may run out of keys.
- If thread A stores a thread-specific data using the **theKey** variable before call 2, the data will be bound to key K1. After call 2, the **theKey** variable contains K2; if thread A then tries to fetch its thread-specific data, it would always get **NULL**.

Ensuring the uniqueness of key creation can be done in two ways:

- Using the one-time initialization facility. See “One-Time Initializations” on page 246.
- Creating the key before the threads that will use it. This is often possible, for example, when using a pool of threads with thread-specific data to perform similar operations. This pool of threads is usually created by one thread, the initial (or another ″driver″) thread.

It is the programmer’s responsibility to ensure the uniqueness of key creation. The threads library provides no way to check if a key has been created more than once.

Destructor Routine

A destructor routine may be associated with each thread-specific data key. Whenever a thread is terminated, if there is non-**NULL,** thread-specific data for this thread bound to any key, the destructor routine associated with that key is called. This allows dynamically allocated thread-specific data to be automatically freed when the thread is terminated. The destructor routine has one parameter, the value of the thread-specific data.

For example, a thread-specific data key may be used for dynamically allocated buffers. A destructor routine should be provided to ensure that the buffer is freed when the thread terminates, the **free** subroutine can be used:

```
pthread_key_create(&key, free);
```

More complex destructors may be used. If a multi-threaded **grep** command, using a thread per file to scan, has thread-specific data to store a structure containing a work buffer and the thread's file descriptor, the destructor routine may be:

```
typedef struct {
        FILE *stream;
        char *buffer;
} data_t;
...
void destructor(void *data)
{
        fclose(((data_t *)data)->stream);
        free(((data_t *)data)->buffer);
        free(data);
        *data = NULL;
}
```

Although some implementations of the threads library may repeat destructor calls, the destructor routine is called only once in AIX. Care must be taken when porting code from other systems where a destructor routine can be called several times.

Key Destruction

A thread-specific data key can be destroyed by calling the **pthread_key_delete** subroutine. This subroutine frees the key only if no thread-specific data is bound to it. Data is said to be bound to the key when at least one value is not **NULL**. The **pthread_key_delete** subroutine does not actually call the destructor routine for each thread having data. To destroy a thread-specific data key, the programmer must ensure that no thread-specific data is bound to the key.

Once a data key is destroyed, it can be reused by another call to the **pthread_key_create** subroutine. Thus, the **pthread_key_delete** is useful especially when using many data keys. For example, in the following code fragment the loop would never end:

```
/* bad example - do not write such code! */
pthread_key_t key;

while (pthread_key_create(&key, NULL))
        pthread_key_delete(key);
```

Using Thread-Specific Data

Thread-specific data is accessed using the **pthread_getspecific** and **pthread_setspecific** subroutines. The first one reads the value bound to the specified key and specific to the calling thread; the second one sets the value.

Setting Successive Values

The value should be a pointer. The pointer may point to any kind of data. Thread-specific data is typically used for dynamically allocated storage, as in the following code fragment:

```
private_data = malloc(...);
pthread_setspecific(key, private_data);
```

When setting a value, the previous value is lost. For example, in the following code fragment, the value of the **old** pointer is lost, and the storage it pointed to may not be recoverable:

```
pthread_setspecific(key, old);
...
pthread_setspecific(key, new);
```

It is the programmer's responsibility to retrieve the old thread-specific data value to reclaim storage before setting the new value. For example, it is possible to implement a **swap_specific** routine in the following manner:

```
int swap_specific(pthread_key_t key, void **old_pt, void *new)
{
        *old_pt = pthread_getspecific(key);
        if (*old_pt == NULL)
                return -1;
        else
                return pthread_setspecific(key, new);
}
```

Such a routine does not exist in the threads library because it is not always necessary to retrieve the previous value of thread-specific data. Such a case occurs, for example, when thread-specific data are pointers to specific locations in a memory pool allocated by the initial thread.

Taking Care about Destructor Routines

When using dynamically allocated thread-specific data, the programmer must provide a destructor routine when calling the **pthread_key_create** subroutine. The programmer must also ensure that, when freeing the storage allocated for thread-specific data, the pointer is set to **NULL**. Otherwise, the destructor routine may be called with an illegal parameter. For example:

```
pthread_key_create(&key, free);
...
...
private_data = malloc(...);
pthread_setspecific(key, private_data);
...
/* bad example! */
...
pthread_getspecific(key, &data);
free(data);
...
```

When the thread terminates, the destructor routine is called for its thread-specific data. Because the value is a pointer to already freed memory, an error may occur. To correct this, the following code fragment should be substituted:

```
/* better example! */
...
pthread_getspecific(key, &data);
free(data);
pthread_setspecific(key, NULL);
...
```

When the thread terminates, the destructor routine is not called, because there is no thread-specific data.

Using Non-Pointer Values

It is possible to store values that are not pointers, such as integers. It is not recommended to do this for at least two reasons:

- Casting a pointer into a scalar type may not be portable
- The **NULL** pointer value is implementation-dependent; several systems assign the **NULL** pointer a non-zero value.

If you are sure that your program will never be ported to another system, you may use integer values for thread-specific data.

Advanced Attributes

This section describes special attributes of threads, mutexes, and condition variables. The implementation of these attributes is optional; it depends on POSIX options. For more information, see “Threads Library Options” on page 261.

Stack Attributes

A stack is allocated for each thread. Stack management is implementation-dependent; thus, the following information applies only to AIX, although similar features may exist on other systems.

The stack is dynamically allocated when the thread is created. Using advanced thread attributes, it is possible for the user to control the “Stack Size” and address of the stack. See “Stack Address” for more information. The following information does not apply to the initial thread, which is created by the system.

Stack Size

The stacksize attribute is defined in AIX. It depends on the “Stack Size POSIX Option” on page 262; this option may not be implemented on other systems.

The stacksize attribute specifies the minimum stack size that will be allocated for a thread. The **pthread_attr_getstacksize** subroutine returns the value of the attribute, and the **pthread_attr_setstacksize** subroutine sets the value.

The stacksize attribute is used as follows to calculate the size of the stack to allocate:

- If the value of stacksize is less than 96KB, a stack of 96KB will be allocated

In the AIX implementation of the threads library, a chunk of data, called *user thread area*, is allocated for each created thread. The allocation is always a multiple of 4KB. The area is divided into:

- A 4KB *red zone*, which is both read and write protected for stack overflow detection
- A cancellation stack with a size equal to User stack size divided by 4 and multiple of 4KB
- A default stack.

> **Note:** The user thread area described here has nothing to do with the **uthread** structure used in the AIX kernel. The user thread area is accessed only in user mode and is exclusively handled by the threads library, whereas the **uthread** structure only exists within the kernel environment.

Stack Address

The stackaddr attribute is not defined in AIX. It depends on the “Stack Address POSIX Option” on page 261; this option is not implemented in AIX but may be implemented on other systems.

The stackaddr attribute specifies the address of the stack that will be allocated for a thread. The **pthread_attr_getstackaddr** subroutine returns the value of the attribute, and the **pthread_attr_setstackaddr** subroutine sets the value.

If no stack address is specified, the stack is allocated by the system at an arbitrary address. There is no way to get this address. Usually you do not need to know the stack address. However, if you really need to have the stack at a known location, you can use the stackaddr attribute. For example, if you need a very large stack, you may set its address to an unused segment, guaranteeing that the allocation will succeed.

If a stack address is specified when calling the **pthread_create** subroutine, the system will try to allocate the stack at the given address. If it fails, the **pthread_create** subroutine returns **EINVAL**. The **pthread_attr_setstackaddr** subroutine never returns an error, unless the specified stack address exceeds the addressing space, because it does not actually allocate the stack.

Process Sharing

Most UNIX systems allow several processes to share a common data space, known as shared memory. AIX also provides this facility; see “Understanding Memory Mapping” on page 537 for more information

about the AIX shared memory facility. The process sharing attributes for condition variables and mutexes are meant to allow these objects to be allocated in shared memory to support synchronization among threads belonging to different processes. However, there is no industry-standard interface for shared memory management. For this reason, the process sharing POSIX option is not implemented in the AIX threads library.

Making Complex Synchronization Objects

The subroutines provided in the threads library can be used as primitives to build more complex synchronization objects. This article provides implementation examples of some traditional synchronization objects:

- "Long Locks", that can be held over long periods of time
- Inter-thread "Semaphores" on page 253
- "Write-Priority Read/Write Locks" on page 254

Long Locks

The mutexes provided by the threads library are low-contention objects and should not be held for a very long time. Long locks are implemented with mutexes and condition variables, so that a long lock may be held for long time without affecting the performance of the program.

The following implementation is very basic. The lock owner is not checked, any thread can unlock any lock. Error handling and cancellation handling are not performed. As written hereafter, long locks should not be used with cancelability enabled. The next example ("Semaphores" on page 253) shows how to prevent data inconsistency using cleanup handlers. However, this example shows a typical use of condition variables.

A long lock has the **long_lock_t** data type. It must be initialized by the **long_lock_init** routine. The **long_lock**, **long_trylock**, and **long_unlock** subroutine performs similar operations to the **pthread_mutex_lock**, **pthread_mutex_trylock**, and **pthread_mutex_unlock** subroutine.

```
typedef struct {
        pthread_mutex_t lock;
        pthread_cond_t cond;
        int free;
        int wanted;
} long_lock_t;

void long_lock_init(long_lock_t *ll)
{
        pthread_mutex_init(&ll->lock, NULL);
        pthread_cond_init(&ll->cond);
        ll->free = 1;
        ll->wanted = 0;
}

void long_lock_destroy(long_lock_t *ll)
{
        pthread_mutex_destroy(&ll->lock);
        pthread_cond_destroy(&ll->cond);
}

void long_lock(long_lock_t *ll)
{
        pthread_mutex_lock(&ll->lock);
        ll->wanted++;
        while(!ll->free)
                pthread_cond_wait(&ll->cond);
        ll->wanted--;
        ll->free = 0;
        pthread_mutex_unlock(&ll->lock);
}
```

```
int long_trylock(long_lock_t *ll)
{
        int got_the_lock;

        pthread_mutex_lock(&ll->lock);
        got_the_lock = ll->free;
        if (got_the_lock)
                ll->free = 0;
        pthread_mutex_unlock(&ll->lock);
        return got_the_lock;
}
void long_unlock(long_lock_t *ll)
{
        pthread_mutex_lock(&ll->lock);
        ll->free = 1;
        if (ll->wanted)
                pthread_cond_signal(&ll->cond);
        pthread_mutex_unlock(&ll->lock);
}
```

Semaphores

Traditional semaphores in UNIX systems are interprocess synchronization facilities. It is possible to implement interthread semaphores for specific usage.

The following implementation is very basic. Error handling is not performed, but cancellations are properly handled with cleanup handlers whenever required.

A semaphore has the **sema_t** data type. It must be initialized by the **sema_init** routine and destroyed with the **sema_destroy** routine. The P and V operations are respectively performed by the **sema_p** and **sema_v** routines.

```
typedef struct {
        pthread_mutex_t lock;
        pthread_cond_t cond;
        int count;
} sema_t;
void sema_init(sema_t *sem)
{
        pthread_mutex_init(&sem->lock, NULL);
        pthread_cond_init(&sem->cond, NULL);
        sem->count = 1;
}
void sema_destroy(sema_t *sem)
{
        pthread_mutex_destroy(&sem->lock);
        pthread_cond_destroy(&sem->cond);
}
void p_operation_cleanup(void *arg)
{
        sema_t *sem;

        sem = (sema_t *)arg;
        pthread_mutex_unlock(&sem->lock);
}
void sema_p(sema_t *sem)
{
        pthread_mutex_lock(&sem->lock);
        pthread_cleanup_push(p_operation_cleanup, sem);
        while (sem->count <= 0)
                pthread_cond_wait(&sem->cond, &sem->lock);
        sem->count--;
        /*
```

```
         *  Note that the pthread_cleanup_pop subroutine will
         *  execute the p_operation_cleanup routine
         */
        pthread_cleanup_pop(1);
}
void sema_v(sema_t *sem)
{
        pthread_mutex_lock(&sem->lock);
        sem->count++;
        if (sem->count <= 0)
                pthread_cond_signal(&sem->cond);
        pthread_mutex_unlock(&sem->lock);
}
```

The counter specifies the number of users that are allowed to take the semaphore. It is never strictly negative; thus, it does not specify the number of waiting users, as for traditional semaphores. This implementation provides a typical solution to the multiple wakeup problem on the **pthread_cond_wait** subroutine. Note that the P operation is cancelable, because the **pthread_cond_wait** subroutine provides a cancellation point.

Write-Priority Read/Write Locks

A write-priority read/write lock provides multiple threads simultaneous read-only access to a protected resource, and a single thread write access to the resource while excluding reads. When a writer releases a lock, other waiting writers will get the lock before any waiting reader. Write-priority read/write locks are usually used to protect resources that are more often read than written.

The following implementation is very basic. The lock owner is not checked, any thread can unlock any lock. Routines similar to the **pthread_mutex_trylock** subroutine are missing and error handling is not performed, but cancellations are properly handled with cleanup handlers whenever required.

A write-priority read/write lock has the **rwlock_t** data type. It must be initialized by the **rwlock_init** routine. The **rwlock_lock_read** routine locks the lock for a reader (multiple readers are allowed), the **rwlock_unlock_read** routine unlocks it. The **rwlock_lock_write** routine locks the lock for a writer, the **rwlock_unlock_write** routine unlocks it. The proper unlocking routine (for the reader or for the writer) must be called.

```
typedef struct {
        pthread_mutex_t lock;
        pthread_cond_t rcond;
        pthread_cond_t wcond;
        int lock_count;  /* < 0 .. held by writer             */
                         /* > 0 .. held by lock_count readers */
                         /* = 0 .. held by nobody             */
        int waiting_writers;  /* count of wating writers      */
} rwlock_t;
void rwlock_init(rwlock_t *rwl)
{
        pthread_mutex_init(&rwl->lock, NULL);
        pthread_cond_init(&rwl->wcond, NULL);
        pthread_cond_init(&rwl->rcond, NULL);
        rwl->lock_count = 0;
        rwl->waiting_writers = 0;
}
void waiting_reader_cleanup(void *arg)
{
        rwlock_t *rwl;

        rwl = (rwlock_t *)arg;
        pthread_mutex_unlock(&rwl->lock);
}
```

```
void rwlock_lock_read(rwlock_t *rwl)
{
        pthread_mutex_lock(&rwl->lock);
        pthread_cleanup_push(waiting_reader_cleanup, rwl);
        while ((rwl->lock_count < 0) && (rwl->waiting_writers))
                pthread_cond_wait(&rwl->rcond, &rwl->lock);
        rwl->lock_count++;
        /*
         *  Note that the pthread_cleanup_pop subroutine will
         *  execute the waiting_reader_cleanup routine
         */
        pthread_cleanup_pop(1);
}
void rwlock_unlock_read(rwlock_t *rwl)
{
        pthread_mutex_lock(&rwl->lock);
        rwl->lock_count--;
        if (!rwl->lock_count)
                pthread_cond_signal(&rwl->wcond);
        pthread_mutex_unlock(&rwl->lock);
}
void waiting_writer_cleanup(void *arg)
{
        rwlock_t *rwl;

        rwl = (rwlock_t *)arg;
        rwl->waiting_writers--;
        if ((!rwl->waiting_writers) && (rwl->lock_count >= 0))
                /*
                         * This only happens if we have been canceled
                         */
                        pthread_cond_broadcast(&rwl->wcond);
                        pthread_mutex_unlock(&rwl->lock);
}
void rwlock_lock_write(rwlock_t *rwl)
{
        pthread_mutex_lock(&rwl->lock);
        rwl->waiting_writers++;
        pthread_cleanup_push(waiting_writer_cleanup, rwl);
        while (rwl->lock_count)
                pthread_cond_wait(&rwl->wcond, &rwl->lock);
        rwl->lock_count = -1;
        /*
         *  Note that the pthread_cleanup_pop subroutine will
         *  execute the waiting_writer_cleanup routine
         */
        pthread_cleanup_pop(1);
}
void rwlock_unlock_write(rwlock_t *rwl)
{
        pthread_mutex_lock(&rwl->lock);
        l->lock_count = 0;
        if (!rwl->wating_writers)
                pthread_cond_broadcast(&rwl->rcond);
        else
                pthread_cond_signal(&rwl->wcond);
        pthread_mutex_unlock(&rwl->lock);
}
```

Readers are just counted. When the count reaches zero, a waiting writer may take the lock. Only one writer can hold the lock. When the lock is released by a writer, another writer is awakened, if there is one. Otherwise, all waiting readers are awakened.

Note that the locking routines are cancelable, because they call the **pthread_cond_wait** subroutine. For this reason, cleanup handlers are registered before calling the subroutine.

List of Threads Advanced-Feature Subroutines

pthread_attr_getstackaddr	Returns the value of the stackaddr attribute of a thread attributes object.
pthread_attr_getstacksize	Returns the value of the stacksize attribute of a thread attributes object.
pthread_attr_setstackaddr	Sets the value of the stackaddr attribute of a thread attributes object.
pthread_attr_setstacksize	Sets the value of the stacksize attribute of a thread attributes object.
pthread_condattr_getpshared	Returns the value of the process-shared attribute of a condition attributes object.
pthread_condattr_setpshared	Sets the value of the process-shared attribute of a condition attributes object.
pthread_getspecific	Returns the thread-specific data associated with the specified key.
pthread_key_create	Creates a thread-specific data key.
pthread_key_delete	Deletes a thread-specific data key.
pthread_mutexattr_getpshared	Returns the value of the process-shared attribute of a mutex attributes object.
pthread_mutexattr_setpshared	Sets the value of the process-shared attribute of a mutex attributes object.
pthread_once	Executes a routine exactly once in a process.
PTHREAD_ONCE_INIT	Initializes a once synchronization control structure.
pthread_setspecific	Sets the thread-specific data associated with the specified key.

Threads-Processes Interactions Overview

Threads and processes interact in several ways, especially when performing the following kind of tasks:

- “Signal Management”
- “Process Duplication and Termination” on page 259
- “Scheduling” on page 260
- “List of Threads-Processes Interactions Subroutines” on page 261.

Signal Management

Signal management in multi-threaded processes resulted from a compromise among many and sometimes conflicting goals. The goal of compatibility is assured: signals in multi-threaded processes are an extension of signals in traditional single-threaded programs. Programs handling signals and written for single-threaded systems will behave as expected in AIX Version 4.

Signal management in multi-threaded processes is shared by the process and thread levels, and consists of:

- Per-process signal handlers
- Per-thread signal masks
- Single delivery of each signal

The threads library also provides a new subroutine and introduces new programming practices for waiting for asynchronously generated signals.

Signal Handlers and Signal Masks

Signal handlers are maintained at process level. It is strongly recommended to use only the **sigaction** subroutine to get and set signal handlers. Other subroutines may not be supported in the future.

Because the list of signal handlers is maintained at process level, any thread within the process may change it. If two threads set a signal handler on the same signal, the last thread that called the **sigaction** subroutine will override the setting of the previous thread call; and in most cases, it will be impossible to predict the order in which threads are scheduled.

Signal masks are maintained at thread level. Each thread can have its own set of signals that will be blocked from delivery. The **sigthreadmask** subroutine must be used to get and set the calling thread's signal mask. The **sigprocmask** subroutine must not be used in multi-threaded programs; otherwise, unexpected behavior may result.

The **sigthreadmask** subroutine is very similar to **sigprocmask**. The parameters and usage of both subroutines are exactly the same. When porting existing code to support the threads library, you may simply replace **sigprocmask** with **sigthreadmask**.

Signal Generation

Signals generated by some action attributable to a particular thread, such as a hardware fault, are sent to the thread that caused the signal to be generated. Signals generated in association with a process ID, a process group ID, or an asynchronous event (such as terminal activity) are sent to the process.

The **pthread_kill** subroutine sends a signal to a thread. Because thread IDs identify threads within a process, this subroutine can only send signals to threads within the same process.

The **kill** subroutine (and thus the **kill** command) sends a signal to a process. A thread can send a signal **Signal** to its process by executing the following call:

```
kill(getpid(), Signal);
```

The **raise** subroutine cannot be used to send a signal to the calling thread's process. The **raise** subroutine sends a signal to the calling thread, as in the following call:

```
pthread_kill(pthread_self(), Signal);
```

This ensures that the signal is sent to the caller of the **raise** subroutine. Thus, library routines written for single-threaded programs may easily be ported to a multi-threaded system, because the **raise** subroutine is usually intended to send the signal to the caller.

The **alarm** subroutine requests that a signal be sent later to the process, and alarm states are maintained at process level. Thus, the last thread that called the **alarm** subroutine overrides the settings of other threads in the process. In a multi-threaded program, the **SIGALRM** signal is not necessarily delivered to the thread that called the **alarm** subroutine. The calling thread may even be terminated; and therefore, it cannot receive the signal.

Handling Signals

Signal handlers are called within the thread to which the signal is delivered. Signal handlers may call the **pthread_self** subroutine to get their thread ID. Some limitations to signal handlers are introduced by the threads library:

- Signal handlers may call the **longjmp** or **siglongjmp** subroutine only if the corresponding call to the **setjmp** or **sigsetjmp** subroutine was performed in the same thread.

Usually, a program that wants to wait for a signal installs a signal handler that calls the **longjmp** subroutine to continue execution at the point where the corresponding **setjmp** subroutine was called. This cannot be done in a multi-threaded program, because the signal may be delivered to a thread other than the one that called the **setjmp** subroutine, thus causing the handler to be executed by the wrong thread.

- Signal handlers must not call the **pthread_cond_signal** or **pthread_cond_broadcast** subroutine to signal a condition.

To allow a thread to wait for asynchronously generated signals, the threads library provides the **sigwait** subroutine. The **sigwait** subroutine blocks the calling thread until one of the awaited signals is sent to the process or to the thread. There must not be a signal handler installed on a signal awaited using the **sigwait** subroutine.

Typically, programs may create a dedicated thread to wait for asynchronously generated signals. Such a thread just loops on a **sigwait** subroutine call and handles the signals. The following code fragment gives an example of such a signal waiter thread:

```
sigset_t set;
int sig;

sigemptyset(&set);
sigaddset(&set, SIGINT);
sigaddset(&set, SIGQUIT);
sigaddset(&set, SIGTERM);
sigthreadmask(SIG_BLOCK, &set, NULL);

while (1) {
        sigwait(&set, &sig);
        switch (sig) {
                case SIGINT:
                        /* handle interrupts */
                        break;
                case SIGQUIT:
                        /* handle quit */
                        break;
                case SIGTERM:
                        /* handle termination */
                        break;
                default:
                        /* unexpected signal */
                        pthread_exit((void *)-1);
        }
}
```

If more than one thread called the **sigwait** subroutine, exactly one call returns when a matching signal is sent. There is no way to predict which thread will be awakened. Note that the **sigwait** subroutine provides a cancellation point.

Because a dedicated thread is not a real signal handler, it may signal a condition (“Using Condition Variables” on page 231) to any other thread . It is possible to implement a **sigwait_multiple** routine that would awaken all threads waiting for a specific signal. Each caller of the **sigwait_multiple** routine would register a set of signals. The caller then waits on a condition variable. A single thread calls the **sigwait** subroutine on the union of all registered signals. When the call to the **sigwait** subroutine returns, the appropriate state is set and condition variables are broadcasted. New callers to the **sigwait_multiple** subroutine would cause the pending **sigwait** subroutine call to be canceled and reissued to update the set of signals being waited for.

Signal Delivery

A signal is delivered to a thread, unless its action is set to ignore. The following rules govern signal delivery in a multi-threaded process:

- A signal whose action is set to terminate, stop, or continue the target thread or process respectively terminates, stops, or continues the entire process (and thus all of its threads). This means that single-threaded programs may be rewritten as multi-threaded programs without changing their externally visible signal behavior.

  Consider for example a multi-threaded user command, such as the **grep** command. A user may start the command in his favorite shell and then decide to stop it by sending a signal with the **kill** command. It is obvious that the signal should stop the entire process running the **grep** command.
- Signals generated for a specific thread, using the **pthread_kill** or the **raise** subroutines, are delivered to that thread. If the thread has blocked the signal from delivery, the signal is set pending on the thread until the signal is unblocked from delivery. If the thread is terminated before the signal delivery, the signal will be ignored.
- Signals generated for a process, using the **kill** subroutine for example, are delivered to exactly one thread in the process. If one or more threads called the **sigwait** subroutine, the signal is delivered to exactly one of these threads. Otherwise, the signal is delivered to exactly one thread that did not block the signal from delivery. If no thread matches these conditions, the signal is set pending on the process until a thread calls the **sigwait** subroutine specifying this signal or a thread unblocks the signal from delivery.

If the action associated with a pending signal (on a thread or on a process) is set to ignore, the signal is ignored.

Process Duplication and Termination

Because all processes have at least one thread, creating (that is, duplicating) and terminating a process implies the creation and the termination of threads. This article describes the interactions between threads and processes when duplicating and terminating a process.

Forking

There are two reasons why AIX programmers call the **fork** subroutine:

1. To create a new flow of control within the same program. AIX creates a new process.
2. To create a new process running a different program. In this case, the call to the **fork** subroutine is soon followed by a call to one of the **exec** subroutines.

In a multi-threaded program, the first use of the **fork** subroutine, creating new flows of control, is provided by the **pthread_create** subroutine. The **fork** subroutine should thus be used only to run new programs.

The **fork** subroutine duplicates the parent process, but duplicates only the calling thread; the child process is a single-threaded process. The calling thread of the parent process becomes the initial thread of the child process; it may not be the initial thread of the parent process. Thus, if the initial thread of the child process returns from its entry-point routine, the child process terminates.

When duplicating the parent process, the **fork** subroutine also duplicates all the synchronization variables, including their state. Thus, for example, mutexes may be held by threads that no longer exist in the child process and any associated resource may be inconsistent.

It is strongly recommended to use the **fork** subroutine only to run new programs, and to call one of the **exec** subroutines as soon as possible after the call to the **fork** subroutine in the child process.

Fork Handlers

Unfortunately, the rule explained above does not address the needs of multi-threaded libraries. Application programs may not be aware that a multi-threaded library is in use and will feel free to call any number of

library routines between the **fork** and the **exec** subroutines, just as they always have. Indeed, they may be old single-threaded programs and cannot, therefore, be expected to obey new restrictions imposed by the threads library.

On the other hand, multi-threaded libraries need a way to protect their internal state during a fork in case a routine is called later in the child process. The problem arises especially in multi-threaded input/output libraries, which are almost sure to be invoked between the **fork** and the **exec** subroutines to affect input/output redirection.

The **pthread_atfork** subroutine provides a way for multi-threaded libraries to protect themselves from innocent application programs which call the **fork** subroutine. It also provides multi-threaded application programs with a standard mechanism for protecting themselves from calls to the **fork** subroutine in a library routine or the application itself.

The **pthread_atfork** subroutine registers fork handlers to be called before and after the call to the **fork** subroutine. The fork handlers are executed in the thread that called the **fork** subroutine. There are three fork handlers:

Prepare	The prepare fork handler is called just before the processing of the **fork** subroutine begins.
Parent	The parent fork handler is called just after the processing of the **fork** subroutine is completed in the parent process.
Child	The child fork handler is called just after the processing of the **fork** subroutine is completed in the child process.

The prepare fork handlers are called in last-in first-out (LIFO) order, whereas the parent and child fork handlers are called in first-in first-out (FIFO) order. This allows programs to preserve any desired locking order.

Process Termination

When a process terminates, by calling the **_exit** subroutine either explicitly or implicitly, all threads within the process are terminated. Neither the cleanup handlers nor the thread-specific data destructors are called.

The reason for this behavior is that there is no state to leave clean and no thread-specific storage to reclaim, because the whole process terminates, including all the threads, and all the process storage is reclaimed, including all thread-specific storage.

Scheduling

In previous versions of AIX, up to AIX 3.2, the process was the schedulable entity. Several commands and subroutines influenced on-process scheduling. The introduction of threads changed the semantics of these commands and subroutines.

Process-Level Scheduling

In AIX Version 4, the scheduler allocates processor time to threads based on each thread's priority and scheduling policy. Previously, AIX scheduled processes and did not support threads: process priority depended on *nice* values, which were managed with the **nice** and **renice** commands and the **getpriority**, **setpriority**, and **nice** subroutines. While these interfaces still exist, process nice values are only used in the default scheduling policy, denoted **SCHED_OTHER**, which uses the nice value and recent CPU usage to calculate priority. The other scheduling policies are fixed priority. The **getpri** and **setpri** subroutines previously managed process priority. They now manage thread priority and respectively return the priority of a thread in the process, or set the priority of all threads in a process.

The **yield** subroutine previously caused a process to relinquish the processor, allowing a higher priority process to be scheduled immediately, before the end of its time slice. In a multi-threaded process, only the calling thread gives up its time slice. Threads can use the **pthread_yield** service to yield the processor. If the contention scope is global, the behavior is as with **yield**, and any thread can be scheduled; if the scope is local, another local thread will be scheduled.

Timer and Sleep Subroutines

Timer routines now execute in the context of the calling thread, instead of the calling process. Thus, if a timer expires, the watchdog timer function is called in the thread's context. When a process or thread goes to sleep, it relinquishes the processor. In a multi-threaded process, only the calling thread is put to sleep.

List of Threads-Processes Interactions Subroutines

alarm	Causes a signal to be sent to the calling process after a specified timeout.
kill or **killpg**	Sends a signal to a process or a group of processes.
pthread_atfork	Registers fork cleanup handlers.
pthread_kill	Sends a signal to the specified thread.
raise	Sends a signal to the executing program.
sigaction, **sigvec**, or **signal**	Specifies the action to take upon delivery of a signal.
sigsuspend or **sigpause**	Atomically changes the set of blocked signals and waits for a signal.
sigthreadmask	Sets the signal mask of a thread.
sigwait	Blocks the calling thread until a specified signal is received.

Threads Library Options

The POSIX standard for the threads library specifies the implementation of some parts as optional. All subroutines defined by the threads library API are always available. Depending on the available options, some subroutines may not be implemented. Unimplemented subroutines can be called by applications, but they always return the **ENOSYS** error code.

List of Options

The following options are defined:

- "Stack Address POSIX Option"
- "Stack Size POSIX Option" on page 262
- "Priority Scheduling POSIX Option" on page 262
- Priority inheritance
- Priority protection
- Process sharing.

The priority inheritance and priority protection options require the implementation of the priority scheduling option.

Stack Address POSIX Option

The stack address option enables the control of the stackaddr attribute of a thread attributes object. This attribute specifies the location of storage to be used for the created thread's stack. See ″"Stack Address" on page 251 for more information about the stackaddr attribute.

The following attribute and subroutines are available when the option is implemented:

- The stackaddr attribute of the thread attributes object

- The **pthread_attr_getstackaddr** and **pthread_attr_setstackaddr** subroutines.

Stack Size POSIX Option

The stack size option enables the control of the stacksize attribute of a thread attributes object. This attribute specifies the minimum stack size to be used for the created thread. See ″“Stack Size” on page 251 for more information about the stacksize attribute.

The following attribute and subroutines are available when the option is implemented:

- The stacksize attribute of the thread attributes object
- The **pthread_attr_getstacksize** and **pthread_attr_setstacksize** subroutines.

Priority Scheduling POSIX Option

The priority scheduling option enables the control of execution scheduling at thread level. When this option is disabled, all threads within a process share the scheduling properties of the process. When this option is enabled, each thread has its own scheduling properties. For local contention scope threads, the scheduling properties are handled at process level by a library scheduler, while for global contention scope threads, the scheduling properties are handled at system level by the kernel scheduler. See “Threads Scheduling” on page 240 to get more information about the scheduling properties of a thread.

The attributes and subroutines shown below are available when the option is implemented:

- The inheritsched attribute of the thread attributes object
- The schedparam attribute of the thread attributes object and the thread
- The schedpolicy attribute of the thread attributes objects and the thread
- The contention-scope attribute of the thread attributes objects and the thread
- The **pthread_attr_getschedparam** and **pthread_attr_setschedparam** subroutines
- The **pthread_getschedparam** subroutine.

Checking the Availability of an Option

Options can be checked at compile time (“Compile Time Checking”) or at run time (“Run Time Checking” on page 263). Portable programs should check the availability of options before using them, so that they do not need to be rewritten when ported to other systems.

Compile Time Checking

Symbolic constants (symbols) can be used to get the availability of options on the system where the program is compiled. The symbols are defined in the **pthread.h** header file by the **#define** pre-processor command. For unimplemented options, the corresponding symbol is undefined by the **#undef** pre-processor command. Checking option symbols should be done in each program that may be ported to another system.

The following list indicates the symbol associated with each option:

Stack address	**_POSIX_THREAD_ATTR_STACKADDR**
Stack size	**_POSIX_THREAD_ATTR_STACKSIZE**
Priority scheduling	**_POSIX_THREAD_PRIORITY_SCHEDULING**
Priority inheritance	**_POSIX_THREAD_PRIO_INHERIT**
Priority protection	**_POSIX_THREAD_PRIO_PROTECT**
Process sharing	**_POSIX_THREAD_PROCESS_SHARED**

The simplest action to take when an option is not available is to stop the compilation, as in the following example:

```
#ifndef _POSIX_THREAD_ATTR_STACKSIZE
#error "The stack size POSIX option is required"
#endif
```

The **pthread.h** header file also defines the following symbols that can be used by other header files or by programs:

_POSIX_REENTRANT_FUNCTIONS	Denotes that reentrant functions are required.
_POSIX_THREADS	Denotes the implementation of the threads library.

Run Time Checking

The **sysconf** subroutine can be used to get the availability of options on the system where the program is executed. This is useful when porting programs between systems that have a binary compatibility, such as two versions of AIX.

The following list indicates the symbolic constant associated with each option and that must be used for the *Name* parameter of the **sysconf** subroutine. The symbolic constants are defined in the **unistd.h** header file.

Stack address	**_SC_THREAD_ATTR_STACKADDR**
Stack size	**_SC_THREAD_ATTR_STACKSIZE**
Priority scheduling	**_SC_THREAD_PRIORITY_SCHEDULING**
Priority inheritance	**_SC_THREAD_PRIO_INHERIT**
Priority protection	**_SC_THREAD_PRIO_PROTECT**
Process sharing	**_SC_THREAD_PROCESS_SHARED**

Two general options may also be checked using the **sysconf** subroutine with the following *Name* parameter values:

_SC_REENTRANT_FUNCTIONS	Denotes that reentrant functions are required.
_SC_THREADS	Denotes the implementation of the threads library.

Threads Library Quick Reference

This section provides a summary of the threads library:

- "Supported Interfaces"
- "Threads Data Types" on page 268
- "Limits and Default Values" on page 269

Supported Interfaces

On AIX systems, _POSIX_THREADS, _POSIX_THREAD_ATTR_STACKADDR, _POSIX_THREAD_ATTR_STACKSIZE and _POSIX_THREAD_PROCESS_SHARED are always defined. Therefore, the following threads interfaces are supported:

POSIX Interfaces

The following is a list of POSIX interfaces:

- pthread_atfork
- pthread_attr_destroy
- pthread_attr_getdetachstate
- pthread_attr_getschedparam
- pthread_attr_getstacksize

- pthread_attr_getstackaddr
- pthread_attr_setdetachstate
- pthread_attr_init
- pthread_attr_setschedparam
- pthread_attr_setstackaddr
- pthread_attr_setstacksize
- pthread_cancel
- pthread_cleanup_pop
- pthread_cleanup_push
- pthread_detach
- pthread_equal
- pthread_exit
- pthread_getspecific
- pthread_join
- pthread_key_create
- pthread_key_delete
- pthread_kill
- pthread_mutex_destroy
- pthread_mutex_init
- pthread_mutex_lock
- pthread_mutex_trylock
- pthread_mutex_unlock
- pthread_mutexattr_destroy
- pthread_mutexattr_getpshared
- pthread_mutexattr_init
- pthread_mutexattr_setpshared
- pthread_once
- pthread_self
- pthread_setcancelstate
- pthread_setcanceltype
- pthread_setspecific
- pthread_sigmask
- pthread_testcancel
- sigwait
- pthread_cond_broadcast
- pthread_cond_destroy
- pthread_cond_init
- pthread_cond_signal
- pthread_cond_timedwait
- pthread_cond_wait
- pthread_condattr_destroy
- pthread_condattr_getpshared
- pthread_condattr_init
- pthread_condattr_setpshared
- pthread_create

Single UNIX Specification Interfaces

The following is a list of single UNIX specification interfaces:

- pthread_attr_getguardsize
- pthread_attr_setguardsize
- pthread_getconcurrency
- pthread_mutexattr_gettype
- pthread_mutexattr_settype
- pthread_rwlock_destroy
- pthread_rwlock_init
- pthread_rwlock_rdlock
- pthread_rwlock_tryrdlock
- pthread_rwlock_trywrlock
- pthread_rwlock_unlock
- pthread_rwlock_wrlock
- pthread_rwlockattr_destroy
- pthread_rwlockattr_getpshared
- pthread_rwlockattr_init
- pthread_rwlockattr_setpshared
- pthread_setconcurrency

On AIX systems, _POSIX_THREAD_SAFE_FUNCTIONS is always defined. Therefore, the following interfaces are always supported:

- asctime_r
- ctime_r
- flockfile
- ftrylockfile
- funlockfile
- getc_unlocked
- getchar_unlocked
- getgrgid_r
- getgrnam_r
- getpwnam_r
- getpwuid_r
- gmtime_r
- localtime_r
- putc_unlocked
- putchar_unlocked
- rand_r
- readdir_r
- strtok_r

AIX does not support the following interfaces; the symbols are provided but they always return an error and set the errno to ENOSYS:

- pthread_attr_getinheritsched
- pthread_attr_getschedpolicy
- pthread_attr_getscope

- pthread_attr_setinheritsched
- pthread_attr_setschedpolicy
- pthread_attr_setscope
- pthread_getschedparam
- pthread_mutex_getprioceiling
- pthread_mutex_setprioceiling
- pthread_mutexattr_getprioceiling
- pthread_mutexattr_getprotocol
- pthread_mutexattr_setprioceiling
- pthread_mutexattr_setprotocol
- pthread_setschedparam

Thread-safety

The following AIX interfaces are not thread-safe.

libc.a Library (Standard Functions):

- advance
- asctime
- brk
- catgets
- chroot
- compile
- ctime
- cuserid
- dbm_clearerr
- dbm_close
- dbm_delete
- dbm_error
- dbm_fetch
- dbm_firstkey
- dbm_nextkey
- dbm_open
- dbm_store
- dirname
- drand48
- ecvt
- encrypt
- endgrent
- endpwent
- endutxent
- fcvt
- gamma
- gcvt
- getc_unlocked
- getchar_unlocked
- getdate

- getdtablesize
- getgrent
- getgrgid
- getgrnam
- getlogin
- getopt
- getpagesize
- getpass
- getpwent
- getpwnam
- getpwuid
- getutxent
- getutxid
- getutxline
- getw
- getw
- gmtime
- l64a
- lgamma
- localtime
- lrand48
- mrand48
- nl_langinfo
- ptsname
- putc_unlocked
- putchar_unlocked
- putenv
- pututxline
- putw
- rand
- random
- readdir
- re_comp
- re_exec
- regcmp
- regex
- sbrk
- setgrent
- setkey
- setpwent
- setutxent
- sigstack
- srand48
- srandom
- step

- strerror
- strtok
- ttyname
- ttyslot
- wait3

libc.a Library (AIX Specific Functions):
- endfsent
- endttyent
- endutent
- getfsent
- getfsfile
- getfsspec
- getfstype
- getttyent
- getttynam
- getutent
- getutid
- getutline
- pututline
- setfsent
- setttyent
- setutent
- utmpname

libbsd.a library:
- timezone

libm.a and **libmsaa.a** libraries:
- gamma
- lgamma

None of the functions in the following libraries are thread safe:
- libPW.a
- libblas.a
- libcur.a
- libcurses.a
- libplot.a
- libprint.a

The interfaces **ctermid** and **tmpnam** are not thread-safe if passed a NULL argument.

> **Note:** Certain subroutines may be implemented as macros on some systems. You should avoid using the address of threads subroutines.

Threads Data Types

The following data types are defined for the threads library in the **pthread.h** header file.

pthread_t
Identifies a thread.

pthread_attr_t
Identifies a thread attributes object.

pthread_cond_t
Identifies a condition variable.

pthread_condattr_t
Identifies a condition attributes object.

pthread_key_t
Identifies a thread-specific data key.

pthread_mutex_t
Identifies a mutex.

pthread_mutexattr_t
Identifies a mutex attributes object.

pthread_once_t
Identifies a one time initialization object.

The definition of these data types can vary from one system to another.

Limits and Default Values

The threads library has some implementation-dependent limits and default values. These limits and default values can be retrieved by symbolic constants to enhance the portability of programs.

Maximum Number of Threads per Process

The maximum number of threads per process is 512. The maximum number of threads can be retrieved at compilation time using the **PTHREAD_THREADS_MAX** symbolic constant defined in the **pthread.h** header file.

Minimum Stack Size

The minimum stack size for a thread is 96KB. It is also the default stack size. This number can be retrieved at compilation time using the **PTHREAD_STACK_MIN** symbolic constant defined in the **pthread.h** header file.

Note that the maximum stack size is 256MB, the size of a segment. This limit is indicated by the **PTHREAD_STACK_MAX** symbolic constant in the **pthread.h** header file.

Maximum Number of Thread-Specific Data Keys

The number of thread-specific data keys is limited to 508. This number can be retrieved at compilation time using the **PTHREAD_KEYS_MAX** symbolic constant defined in the **pthread.h** header file.

Default Attribute Values

The default values for the thread attributes object are defined in the **pthread.h** header file by the following symbolic constants:

DEFAULT_DETACHSTATE
PTHREAD_CREATE_DETACHED. Specifies the default value for the detachstate attribute.

DEFAULT_INHERIT
PTHREAD_INHERIT_SCHED. Specifies the default value for the inheritsched attribute.

DEFAULT_PRIO
1 (one). Specifies the default value for the `sched_prio` field of the schedparam attribute.

DEFAULT_SCHED
SCHED_OTHER. Specifies the default value for the schedpolicy attribute of a thread attributes object.

DEFAULT_SCOPE
PTHREAD_SCOPE_LOCAL. Specifies the default value for the contention-scope attribute.

The actual values shown might vary from one release to another.

Chapter 12. lex and yacc Program Information

The **lex** command generates program that matches patterns for simple lexical analysis of an input stream. The **yacc** command converts a context-free grammar specification into a set of tables for a simple automaton that executes a parser. Together these commands generate a lexical analyzer and parser program for interpreting input and output handling.

The following topics are covered in this chapter:

"Creating an Input Language with the lex and yacc Commands"

"Example Program for the lex and yacc Programs" on page 296

Creating an Input Language with the lex and yacc Commands

lex Generates a lexical analyzer program that analyzes input and breaks it into tokens, such as numbers, letters, or operators. The tokens are defined by grammar rules set up in the **lex** specification file.

yacc Generates a parser program that analyzes input using the tokens identified by the lexical analyzer (generated by the **lex** command and stored in the **lex** specification file) and performs specified actions, such as flagging improper syntax.

Writing a Lexical Analyzer Program with the lex Command

The **lex** command helps write a C language program that can receive and translate character-stream input into program actions. To use the **lex** command, you must supply or write a specification file that contains:

Extended regular expressions	Character patterns that the generated lexical analyzer recognizes.
Action statements	C language program fragments that define how the generated lexical analyzer reacts to extended regular expressions it recognizes.

The format and logic allowed in this file are discussed in the **lex** Specification File section of the **lex** command.

How the lex Command Operates

The **lex** command generates a C language program that can analyze an input stream using information in the specification file. The **lex** command then stores the output program in a **lex.yy.c** file. If the output program recognizes a simple, one-word input structure, you can compile the **lex.yy.c** output file with the following command to get an executable lexical analyzer:

```
cc lex.yy.c -ll
```

However, if the lexical analyzer must recognize more complex syntax, you can create a parser program to use with the output file to ensure proper handling of any input. See "Creating a Parser with the yacc Program" on page 283 for more information.

You can move a **lex.yy.c** output file to another system if it has C compiler that supports the **lex** library functions.

The compiled lexical analyzer performs the following functions:

- Reads an input stream of characters.
- Copies the input stream to an output stream.

- Breaks the input stream into smaller strings that match the extended regular expressions in the **lex** specification file.
- Executes an action for each extended regular expression that it recognizes. These actions are C language program fragments in the **lex** specification file. Each action fragment can call actions or subroutines outside of itself.

How the Lexical Analyzer Works

The lexical analyzer that the **lex** command generates uses an analysis method called a *deterministic finite-state automaton*. This method provides for a limited number of conditions that the lexical analyzer can exist in, along with the rules that determine what state the lexical analyzer is in.

The automaton allows the generated lexical analyzer to look ahead more than one or two characters in an input stream. For example, suppose you define two rules in the **lex** specification file: one looks for the string `ab` and the other looks for the string `abcdefg`. If the lexical analyzer receives an input string of `abcdefh`, it reads characters to the end of input string before determining that it does not match the string `abcdefg`. The lexical analyzer then returns to the rule that looks for the string `ab`, decides that it matches part of the input, and begins trying to find another match using the remaining input `cdefh`.

Extended Regular Expressions in the lex Command

Specifying extended regular expressions in a **lex** specification file is similar to methods used in the **sed** or **ed** commands. An extended regular expression specifies a set of strings to be matched. The expression contains both text characters and operator characters. Text characters match the corresponding characters in the strings being compared. Operator characters specify repetitions, choices, and other features.

Numbers and letters of the alphabet are considered text characters. For example, the extended regular expression `integer` matches the string `integer`, and the expression `a57D` looks for the string `a57D`.

Operators

The following list describes how operators are used to specify extended regular expressions:

Expression	Use
Character	Matches the character *Character.* **Example:** `a` matches the literal character a; `b` matches the literal character b, and `c` matches the literal character c.
"String"	Matches the string enclosed within quotes, even if the string includes an operator. **Example:** to prevent the **lex** command from interpreting `$` (dollar sign) as an operator, enclose the symbol in quotes.

Character or *Digits*

Escape character. When preceding a character class operator used in a string, the \ character indicates that the operator symbol represents a literal character rather than an operator. Valid escape sequences include:

\a Alert

\b Backspace

\f Form-feed

\n New-line character (Do not use the actual new-line character in an expression.)

\r Return

\t Tab

\v Verticle tab

**** Backslash

*****Digits* The character whose encoding is represented by the one-, two-, or three-digit octal integer specified by the *Digits* string.

\x*Digits*
The character whose encoding is represented by the sequence of hexadecimal characters specified by the *Digits* string.

When the \ character precedes a character that is not in the preceding list of escape sequences, the **lex** command interprets the character literally.

Example: `\c` is interpreted as the `c` character unchanged, and `[\^abc]` represents the class of characters that includes the characters `^abc`.

Note: Never use `\0` or `\x0` in lex rules.

[*List*]

Matches any one character in the enclosed range ([*x-y*]) or the enclosed list ([*xyz*]) based on the locale in which the **lex** command is invoked. All operator symbols, with the exception of the following, lose their special meaning within a bracket expression: - (dash), ^ (carat), and \ (backslash).

Example: `[abc-f]` matches a, b, c, d, e, or f in the En_US locale.

[:*Class*:]
Matches any of the characters belonging to the character class specified between the [::] delimiters as defined in the LC_TYPE category in the current locale. The following character class names are supported in all locales:

alnum cntrl lower space

alpha digit print upper

blank graph punct xdigit

The **lex** command also recognizes user-defined character class names. The [::] operator is valid only in a [] expression.

Example: [[:alpha:]] matches any character in the **alpha** character class in the current locale, but [:alpha:] matches only the characters :,a,l,p, and h.

[.*CollatingSymbol*.]
Matches the collating symbol specified within the [..] delimiters as a single character. The [..] operator is valid only in a [ ] expression. The collating symbol must be a valid collating symbol for the current locale.

Example: [[.ch.]] matches c and h together while [ch] matches c or h.

[=*CollatingElement*=]
Matches the collating element specified within the [==] delimiters and all collating elements belonging to its equivalence class. The [==] operator is valid only in a [] expression.

Example: If w and v belong to the same equivalence class, [[=w=]] is the same as [wv] and matches w or v. If w does not belong to an equivalence class, then [[=w=]] matches w only.

[^*Character*]
Matches any character except the one following the ^ (caret) symbol.The resultant character class consists solely of single-byte characters. The character following the ^ symbol can be a multibyte character, however for this operator to match multibyte characters, you must set **%h** and **%m** to greater than zero in the definitions section.

Example: [^c] matches any character except c.

CollatingElement-*CollatingElement*
In a character class, indicates a range of characters within the collating sequence defined for the current locale. Ranges must be in ascending order. The ending range point must collate equal to or higher than the starting range point. Because the range is based on the collating sequence of the current locale, a given range may match different characters, depending on the locale in which the **lex** command was invoked.

Expression?
Matches either zero or one occurrence of the expression immediately preceding the ? operator.

Example: ab?c matches either ac or abc.

.	Matches any character except the new-line character. In order for **.** to match multi-byte characters, **%z** must be set to greater than 0 in the definitions section of the **lex** specification file. If **%z** is not set, . matches single-byte characters only.
*Expression**	Matches zero or more occurrences of the expression immediately preceding the * operator. For example, a* is any number of consecutive a characters, including zero. The usefulness of matching zero occurrences is more obvious in complicated expressions. **Example:** The expression, [A-Za-z][A-Za-z0-9]* indicates all alphanumeric strings with a leading alphabetic character, including strings that are only one alphabetic character. You can use this expression for recognizing identifiers in computer languages.
Expression+	Matches one or more occurrences of the pattern immediately preceding the + operator. **Example:** a+ matches one or more instances of a. Also, [a-z]+ matches all strings of lowercase letters.
Expression|*Expression*	Indicates a match for the expression that precedes or follows the | (pipe) operator. **Example:** ab|cd matches either ab or cd.
(*Expression*)	Matches the expression in the parentheses. The () (parentheses) operator is used for grouping and causes the expression within parentheses to be read into the **yytext** array. A group in parentheses can be used in place of any single character in any other pattern. **Example:** (ab|cd+)?(ef)* matches such strings as abefef, efefef, cdef, or cddd; but not abc, abcd, or abcdef.
^*Expression*	Indicates a match only when the ^ (carat) operator is at the beginning of the line and the ^ is the first character in an expression. **Example:** ^h matches an h at the beginning of a line.
Expression$	Indicates a match only when the $ (dollar sign) is at the end of the line and the $ is the last character in an expression. **Example:** The description of *Expression*/*Expression*.

Expression1/*Expression2*	Indicates a match only if *Expression2* immediately follows *Expression1*. The / (slash) operator reads only the first expression into the **yytext** array. **Example:** `ab/cd` matches the string `ab`, but only if followed by `cd`, and then reads `ab` into the **yytext** array. **Note:** Only one / trailing context operator can be used in a single extended regular expression. The ^ (carat) and $ (dollar sign) operators cannot be used in the same expression with the / operator as they indicate special cases of trailing context.
{*DefinedName*}	Matches the name as you defined it in the definitions section. **Example:** If you defined `D` to be numerical digits, `{D}` matches all numerical digits.
{*Number1*,*Number2*}	Matches *Number1* to *Number2* occurrences of the pattern immediately preceding it. The expressions {*Number*} and {*Number*,} are also allowed and match exactly *Number* occurrences of the pattern preceding the expression. **Example:** `xyz{2,4}` matches either xyzxyz, xyzxyzxyz, or xyzxyzxyzxyz. This differs from the +, * and ? operators in that these operators match only the character immediately preceding them. To match only the character preceding the interval expression, use the grouping operator. For example, `xy(z{2,4})` matches xyzz, xyzzz or xyzzzz.
<*StartCondition*>	Executes the associated action only if the lexical analyzer is in the indicated start condition (“lex Start Conditions” on page 281). **Example:** If being at the beginning of a line is start condition `ONE`, then the ^ (caret) operator equals the expression, `<ONE>`.

To use the operator characters as text characters, use one of the escape sequences: " " (double quotation marks) or \ (backslash). The " " operator indicates what is enclosed is text. Thus, the following example matches the string `xyz++`:

```
xyz"++"
```

Note that a part of a string may be quoted. Quoting an ordinary text character has no effect. For example, the following expression is equivalent to the previous example:

```
"xyz++"
```

Quoting all characters that are not letters or numbers ensures that text is interpreted as text.

Another way to turn an operator character into a text character is to put a \ (backslash) character before the operator character. For example, the following expression is equivalent to the preceding examples:

```
xyz\+\+
```

lex Actions

When the lexical analyzer matches one of the extended regular expressions in the rules section of the specification file, it executes the *action* that corresponds to the extended regular expression. Without sufficient rules to match all strings in the input stream, the lexical analyzer copies the input to standard output. Therefore, do not create a rule that only copies the input to the output. The default output can help find gaps in the rules.

When using the **lex** command to process input for a parser that the **yacc** command produces, provide rules to match all input strings. Those rules must generate output that the **yacc** command can interpret.

Null Action

To ignore the input associated with an extended regular expression, use a ; (C language null statement) as an action. The following example ignores the three spacing characters (blank, tab, and new-line):

```
[ \t\n] ;
```

Same As Next Action

To avoid repeatedly writing the same action, use the | (pipe symbol). This character indicates that the action for this rule is the same as the action for the next rule. For instance, the previous example to ignore blank, tab, and new-line characters can also be written as follows:

```
" "                     |
"\t"                    |
"\n"                    ;
```

The quotation marks around `\n` and `\t` are not required.

Printing a Matched String

To find out what text matched an expression in the rules section of the specification file, you can include a C language **printf** subroutine call as one of the actions for that expression. When the lexical analyzer finds a match in the input stream, the program puts the matched string into the external character (**char**) and wide character (**wchar_t**) arrays, called **yytext** and **yywtext,** respectively. For example, you can use the following rule to print the matched string:

```
[a-z]+            printf("%s",yytext);
```

The C language **printf** subroutine accepts a format argument and data to be printed. In this example the arguments to the **printf** subroutine have the following meanings:

%s	A symbol that converts the data to type string before printing
%S	A symbol that converts the data to wide character string (**wchar_t**) before printing
yytext	The name of the array containing the data to be printed
yywtext	The name of the array containing the multibyte type (**wchar_t**) data to be printed.

The **lex** command defines **ECHO**; as a special action to print out the contents of **yytext**. For example, the following two rules are equilvalent:

```
[a-z]+       ECHO;
[a-z]+       printf("%s",yytext);
```

You can change the representation of **yytext** by using either **%array** or **%pointer** in the definitions section of the **lex** specification file.

%array	Defines **yytext** as a null-terminated character array. This is the default action.
%pointer	Defines **yytext** as a pointer to a null-terminated character string.

Finding the Length of a Matched String

To find the number of characters that the lexical analyzer matched for a particular extended regular expression, use the **yyleng** or the **yywleng** external variables.

yyleng Tracks the number of bytes that are matched.
yywleng Tracks the number of wide characters in the matched string. Multibyte characters have a length greater than 1.

To count both the number of words and the number of characters in words in the input, use the following action:

```
[a-zA-Z]+       {words++;chars += yyleng;}
```

This action totals the number of characters in the words matched and puts that number in `chars`.

The following expression finds the last character in the string matched:

```
yytext[yyleng-1]
```

Matching Strings within Strings

The **lex** command partitions the input stream and does not search for all possible matches of each expression. Each character is accounted for only once. To override this choice and search for items that may overlap or include each other, use the **REJECT** directive. For example, to count all instance of `she` and `he`, including the instances of `he` that are included in `she`, use the following action:

```
she             {s++; REJECT;}
he              {h++}
\n              |
.               ;
```

After counting the occurrences of `she`, the **lex** command rejects the input string and then counts the occurrences of `he`. Because `he` does not include `she`, a **REJECT** action is not necessary on `he`.

Getting More Input

Normally, the next string from the input stream overwrites the current entry in the **yytext** array. If you use the **yymore** subroutine, the next string from the input stream is added to the end of the current entry in the **yytext** array.

For example, the following lexical analyser looks for strings:

```
%s instring
%%
<INITIAL>\"     {  /* start of string */
         BEGIN instring;
         yymore();
        }
<instring>\"    {  /* end of string */
         printf("matched %s\n", yytext);
         BEGIN INITIAL;
        }
<instring>.     {
         yymore();
        }
<instring>\n    {
         printf("Error, new line in string\n");
         BEGIN INITIAL;
        }
```

Even though a string may be recognized by matching several rules, repeated calls to the **yymore** subroutine ensure that the **yytext** array will contain the entire string.

Putting Characters Back

To return characters to the input stream, use the call:

```
yyless(n)
```

where n is the number of characters of the current string to keep. Characters in the string beyond this number are returned to the input stream. The **yyless** subroutine provides the same type of look-ahead that the / (slash) operator uses, but it allows more control over its usage.

Use the **yyless** subroutine to process text more than once. For example, when parsing a C language program, an expression such asx=-a is difficult to understand. Does it mean x *is equal to minus* a, or is it an older representation of x -= a which means *decrease* x *by the value of* a? To treat this expression as x *is equal to minus* a, but print a warning message, use a rule such as:

```
=-[a-zA-Z]      {
                printf("Operator (=-) ambiguous\n");
                yyless(yyleng-1);
                ... action for = ...
                }
```

Input/Output Subroutines

The **lex** program allows a program to use the following input/output (I/O) subroutines:

input()	Returns the next input character.
output(c)	Writes the character c on the output.
unput(c)	Pushes the character c back onto the input stream to be read later by the **input** subroutine.
winput()	Returns the next multibyte input character.
woutput(C)	Writes the multibyte character C back onto the output stream.
wunput(C)	Pushes the multibyte character C back onto the input stream to be read by the **winput** subroutine.

lex provides these subroutines as macro definitions. The subroutines are coded in the **lex.yy.c** file. You can override them and provide other versions.

The **winput**, **wunput**, and **woutput** macros are defined to use the **yywinput**, **yywunput**, and **yywoutput** subroutines. For compatibility, the **yy** subroutines subsequently use the **input**, **unput**, and **output** subroutine to read, replace, and write the necessary number of bytes in a complete multibyte character.

These subroutines define the relationship between external files and internal characters. If you change the subroutines, change them all in the same way. They should follow these rules:

- All subroutines must use the same character set.
- The **input** subroutine must return a value of 0 to indicate end of file.
- Do not change the relationship of the **unput** subroutine to the **input** subroutine or the look-ahead functions will not work.

The **lex.yy.c** file allows the lexical analyzer to back up a maximum of 200 characters.

To read a file containing nulls, create a different version of the **input** subroutine. In the normal version of the **input** subroutine, the returned value of 0 (from the null characters) indicates the end of file and ends the input.

Character Set

The lexical analyzers that the **lex** command generates process character I/O through the **input, output,** and **unput** subroutines. Therefore, to return values in the **yytext** subroutine, the **lex** command uses the character representation that these subroutines use. Internally however, the **lex** command represents each character with a small integer. When using the standard library, this integer is the value of the bit pattern the computer uses to represent the character. Normally, the letter 'a' is represented in the same form as

the character constant 'a'. If you change this interpretation with different I/O subroutines, put a translation table in the definitions section of the specification file. The translation table begins and ends with lines that contain only the entries:

```
%T
```

The translation table contains additional lines that indicate the value associated with each character. For example:

```
%T
{integer}       {character string}
{integer}       {character string}
{integer}       {character string}
%T
```

End-of-File Processing

When the lexical analyzer reaches the end of a file, it calls the **yywrap** library subroutine.

yywrap Returns a value of 1 to indicate to the lexical analyzer that it should continue with normal wrap-up at the end of input

However, if the lexical analyzer receives input from more than one source, change the **yywrap** subroutine. The new function must get the new input and return a value of 0 to the lexical analyzer. A return value of 0 indicates the program should continue processing.

You can also include code to print summary reports and tables when the lexical analyzer ends in a new version of the **yywrap** subroutine. The **yywrap** subroutine is the only way to force the **yylex** subroutine to recognize the end of input.

Passing Code to the Generated lex Program

The **lex** command passes C code, unchanged, to the lexical analyzer in the following circumstances:

- Lines beginning with a blank or tab in the definitions section, or at the start of the rules section before the first rule, are copied into the lexical analyzer. If the entry is in the definitions section, it is copied to the external declaration area of the **lex.yy.c** file. If the entry is at the start of the rules section, the entry is copied to the local declaration area of the **yylex** subroutine in the **lex.yy.c** file.
- Lines that lie between delimiter lines containing only `%{` (percent sign, left brace) and `%}` (percent sign, right brace) either in the definitions section or at the start of the rules section are copied into the lexical analyzer in the same way as lines beginning with a blank or tab.
- Any lines occurring after the second `%%` (percent sign, percent sign) delimiter are copied to the lexical analyzer without format restrictions.

Defining lex Substitution Strings

You can define string macros that the **lex** program expands when it generates the lexical analyzer. Define them before the first `%%` delimiter in the **lex** specification file. Any line in this section that begins in column 1 and that does not lie between `%{` and `%}` defines a **lex** substitution string. Substitution string definitions have the general format:

```
name                    translation
```

where `name` and `translation` are separated by at least one blank or tab, and the specified name begins with a letter. When the **lex** program finds the string defined by `name` enclosed in `{}` (braces) in the rules part of the specification file, it changes that name to the string defined in `translation` and deletes the braces.

For example, to define the names D and E, put the following definitions before the first %% delimiter in the specification file:

```
D           [0-9]
E           [DEde][-+]{D}+
```

Then, use these names in the rules section of the specification file to make the rules shorter:

```
{D}+                        printf("integer");
{D}+"."{D}*({E})?           |
{D}*"."{D}+({E})?           |
{D}+{E}                     printf("real");
```

You can also include the following items in the definitions section:

- Character set table
- List of start conditions
- Changes to size of arrays to accommodate larger source programs

lex Start Conditions

A rule may be associated with any start condition. However, the **lex** program recognizes the rule only when in that associated start condition. You can change the current start condition at any time.

Define start conditions in the *definitions* section of the specification file by using a line in the following form:

```
%Start  name1 name2
```

where name1 and name2 define names that represent conditions. There is no limit to the number of conditions, and they can appear in any order. You can also shorten the word Start to s or S.

When using a start condition in the rules section of the specification file, enclose the name of the start condition in <> (less than, greater than) symbols at the beginning of the rule. The following example defines a rule, expression, that the **lex** program recognizes only when the **lex** program is in start condition name1:

```
<name1>expression
```

To put the **lex** program in a particular start condition, execute the action statement in the action part of a rule; for instance, BEGIN in the following line:

```
BEGIN name1;
```

This statement changes the start condition to name1.

To resume the normal state, enter:

```
BEGIN 0;
```

or

```
BEGIN INITIAL;
```

where INITIAL is defined to be 0 by the **lex** program. BEGIN 0; resets the **lex** program to its initial condition.

The **lex** program also supports exclusive start conditions specified with %**x** (percent sign, lowercase x) or %**X** (percent sign, uppercase X) operator followed by a list of exclusive start names in the same format as regular start conditions. Exclusive start conditions differ from regular start conditions in that rules that do not begin with a start condition are not active when the lexical analyzer is in an exclusive start state. For example:

```
%s      one
%x      two
%%
abc     {printf("matched ");ECHO;BEGIN one;}
<one>def         printf("matched ");ECHO;BEGIN two;}
<two>ghi         {printf("matched ");ECHO;BEGIN INITIAL;}
```

In start state one in the preceding example, both `abc` and `def` can be matched. In start state two, only `ghi` can be matched.

Compiling the Lexical Analyzer

Compiling a **lex** program is a two-step process:

1. Use the **lex** program to change the specification file into a C language program. The resulting program is in the **lex.yy.c** file.
2. Use the **cc** command with the **-ll** flag to compile and link the program with a library of **lex** subroutines. The resulting executable program is in the **a.out** file.

For example, if the **lex** specification file is called **lextest**, enter the following commands:

```
lex lextest
cc lex.yy.c -ll
```

lex Library

The **lex** library contains the following subroutines:

main()	Invokes the lexical analyser by calling the **yylex** subroutine.
yywrap()	Returns the value 1 when the end of input occurs.
yymore()	Appends the next matched string to the current value of the **yytext** array rather than replacing the contents of the **yytext** array.
yyless(int *n*)	Retains *n* initial characters in the **yytext** array and returns the remaining characters to the input stream.
yyreject()	Allows the lexical analyser to match multiple rules for the same input string. (**yyreject** is called when the special action **REJECT** is used.)

Some of the **lex** subroutines can be substituted by user-supplied routines. For example, **lex** supports user-supplied versions of the **main** and **yywrap** subroutines. The library versions of these routines, provided as a base, are as follows:

main

```
#include <stdio.h>
#include <locale.h>
main() {
     setlocale(LC_ALL, "");
     yylex();
     exit(0);
}
```

yywrap

```
yywrap() {
       return(1);
}
```

yymore, **yyless**, and **yyreject** subroutines are available only through the **lex** library; however, these subroutines are required only when used in **lex** actions.

Using the lex Program with the yacc Program

When used alone, the **lex** program generator makes a lexical analyzer that recognizes simple, one-word input or receives statistical input. You can also use the **lex** program with a parser generator, such as the **yacc** command. The **yacc** command generates a program, called a parser, that analyzes the construction of more than one-word input. This parser program operates well with the lexical analyzers that the **lex** command generates. The parsers recognize many types of grammar with no regard to context. These parsers need a preprocessor to recognize input tokens such as the preprocessor that the **lex** command produces.

The **lex** program recognizes only extended regular expressions and formats them into character packages called tokens, as specified by the input file. When using the **lex** program to make a lexical analyzer for a parser, the lexical analyzer (created from the **lex** command) partitions the input stream. The parser (from the **yacc** command) assigns structure to the resulting pieces. You can also use other programs along with the programs generated by either the **lex** or **yacc** commands.

A token is the smallest independent unit of meaning as defined by either the parser or the lexical analyzer. A token can contain data, a language keyword, an identifier, or other parts of a language syntax.

The **yacc** program looks for a lexical analyzer subroutine named **yylex**, which is generated by the **lex** command. Normally, the default main program in the **lex** library calls the **yylex** subroutine. However, if the **yacc** command is loaded and its main program is used, **yacc** calls the **yylex** subroutine. In this case, each **lex** rule should end with:

```
return(token);
```

where the appropriate `token` value is returned.

The **yacc** command assigns an integer value to each token defined in the **yacc** grammar file through a #**define** preprocessor statement. The lexical analyzer must have access to these macros to return the tokens to the parser. Use the **yacc -d** option to create a **y.tab.h** file, and include the **y.tab.h** file in the **lex** specification file by adding the following lines to the definition section of the **lex** specification file:

```
%{
#include "y.tab.h"
%}
```

Alternately, you can include the **lex.yy.c** file in the **yacc** output file by adding the following line after the second `%%` (percent sign, percent sign) delimiter in the **yacc** grammar file:

```
#include "lex.yy.c"
```

The **yacc** library should be loaded before the **lex** library to get a main program that invokes the **yacc** parser. You can generate **lex** and **yacc** programs in either order.

Creating a Parser with the yacc Program

The **yacc** program creates parsers that define and enforce structure for character input to a computer program. To use this program, you must supply the following inputs:

Grammar file	A source file that contains the specifications for the language to recognize. This file also contains the **main**, **yyerror**, and **yylex** subroutines. You must supply these subroutines.
main	A C language subroutine that, as a minimum, contains a call to the **yyparse** subroutine generated by the **yacc** program. A limited form of this subroutine is available in the **yacc** library.
yyerror	A C language subroutine to handle errors that can occur during parser operation. A limited form of this subroutine is available in the **yacc** library.

yylex A C language subroutine to perform lexical analysis on the input stream and pass tokens to the parser. You can generate this lexical analyzer subroutine using the **lex** command.

When the **yacc** command gets a specification, it generates a file of C language functions called **y.tab.c**. When compiled using the **cc** command, these functions form the **yyparse** subroutine and return an integer. When called, the **yyparse** subroutine calls the **yylex** subroutine to get input tokens. The **yylex** subroutine continues providing input until either the parser detects an error or the **yylex** subroutine returns an end-marker token to indicate the end of operation. If an error occurs and the **yyparse** subroutine cannot recover, it returns a value of 1 to **main**. If it finds the end-marker token, the **yyparse** subroutine returns a value of 0 to **main**.

yacc Grammar File

To use the **yacc** command to generate a parser, give it a grammar file that describes the input data stream and what the parser is to do with the data. The grammar file includes rules describing the input structure, code to be invoked when these rules are recognized, and a subroutine to do the basic input.

The **yacc** command uses the information in the grammar file to generate a parser that controls the input process. This parser calls an input subroutine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. The parser organizes these tokens according to the structure rules in the grammar file. The structure rules are called grammar rules. When the parser recognizes one of these rules, it executes the user code supplied for that rule. The user code is called an action. Actions return values and use the values returned by other actions.

Use the C programming language to write the action code and other subroutines. The **yacc** command uses many of the C language syntax conventions for the grammar file.

main and yyerror Subroutines

You must provide the **main** and **yyerror** subroutines for the parser. To ease the initial effort of using the **yacc** command, the **yacc** library contains simple versions of the **main** and **yyerror** subroutines. Include these subroutines by using the *-ly* argument to the **ld** command (or to the **cc** command). The source code for the **main** library program is:

```
#include <locale.h>
main()
{
     setlocale(LC_ALL, "");
     yyparse();
}
```

The source code for the **yyerror** library program is:

```
#include <stdio.h>
yyerror(s)
        char *s;
{
        fprintf( stderr, "%s\n" ,s);
}
```

The argument to the **yyerror** subroutine is a string containing an error message, usually the string `syntax error`.

These are very limited programs. You should provide more function in these subroutines. For example, keep track of the input line number and print it along with the message when a syntax error is detected. You may also want to use the value in the external integer variable **yychar**. This variable contains the look-ahead token number at the time the error was detected.

yylex Subroutine

The input subroutine that you supply must be able to:

- Read the input stream.
- Recognize basic patterns in the input stream.
- Pass the patterns to the parser along with tokens that define the pattern to the parser.

A token is a symbol or name that tells the parser which pattern is being sent to it by the input subroutine. A nonterminal symbol is the structure that the parser recognizes.

For example, if the input subroutine separates an input stream into the tokens of `WORD`, `NUMBER`, and `PUNCTUATION`, and it receives the following input:

```
I have 9 turkeys.
```

the program could choose to pass the following strings and tokens to the parser:

String	Token
`I`	WORD
`have`	WORD
`9`	NUMBER
`turkeys`	WORD
`.`	PUNCTUATION

The parser must contain definitions for the tokens passed to it by the input subroutine. Using the **-d** option for the **yacc** command, it generates a list of tokens in a file called **y.tab.h**. This list is a set of **#define** statements that allow the lexical analyzer (**yylex**) to use the same tokens as the parser.

To avoid conflict with the parser, do not use names that begin with the letters yy.

You can use the **lex** command to generate the input subroutine or you can write the routine in the C language.

Using the yacc Grammar File

A **yacc** grammar file consists of three sections:

- Declarations
- Rules
- Programs

Two adjacent `%%` (double percent signs) separate each section of the grammar file. To make the file easier to read, put the `%%` on a line by themselves. A complete grammar file looks like:

```
declarations
%%
rules
%%
programs
```

The declarations section may be empty. If you omit the programs section, omit the second set of `%%`. Therefore, the smallest **yacc** grammar file is:

```
%%
rules
```

The **yacc** command ignores blanks, tabs and new-line characters in the grammar file. Therefore, use these characters to make the grammar file easier to read. Do not, however, use blanks, tabs or new-lines in names or reserved symbols.

Using Comments
Put comments in the grammar file to explain what the program is doing. You can put comments anywhere in the grammar file you can put a name. However, to make the file easier to read, put the comments on lines by themselves at the beginning of functional blocks of rules. A comment in a **yacc** grammar file looks the same as a comment in a C language program. The comment is enclosed between /* (backslash, asterisk) and */ (asterisk, backslash). For example:

```
/* This is a comment on a line by itself. */
```

Using Literal Strings
A literal string is one or more characters enclosed in '' (single quotes). As in the C language, the \ (backslash) is an escape character within literals, and all the C language escape codes are recognized. Thus, the **yacc** command accepts the symbols in the following table:

Symbol	Definition
'**a**'	Alert
'**n**'	New-line
'**r**'	Return
'\''	Single quote (')
'\"'	Double quote (")
'\?'	Question mark (?)
'\\'	Backslash (\)
'**t**'	Tab
'**v**'	Vertical tab
'**b**'	Backspace
'**f**'	Form-feed
'*Digits*'	The character whose encoding is represented by the one-, two-, or three-digit octal integer specified by the *Digits* string.
'**x***Digits*'	The character whose encoding is represented by the sequence of hexadecimal characters specified by the *Digits* string.

Never use \0 or 0 (the null character) in grammar rules.

Formatting the Grammar File
Use the following guidelines to help make the **yacc** grammar file more readable:

- Use uppercase letters for token names and lowercase letters for nonterminal symbol names.
- Put grammar rules and actions on separate lines to allow changing either one without changing the other.
- Put all rules with the same left side together. Enter the left side once, and use the vertical bar to begin the rest of the rules for that left side.
- For each set of rules with the same left side, enter the semicolon once on a line by itself following the last rule for that left side. You can then add new rules easily.
- Indent rule bodies by two tab stops and action bodies by three tab stops.

Errors in the Grammar File
The **yacc** command cannot produce a parser for all sets of grammar specifications. If the grammar rules contradict themselves or require matching techniques that are different from what the **yacc** command provides, the **yacc** command will not produce a parser. In most cases, the **yacc** command provides messages to indicate the errors. To correct these errors, redesign the rules in the grammar file, or provide a lexical analyzer (input program to the parser) to recognize the patterns that the **yacc** command cannot.

yacc Declarations

The declarations section of the **yacc** grammar file contains:

- Declarations for any variables or constants used in other parts of the grammar file

- #**include** statements to use other files as part of this file (used for library header files)
- Statements that define processing conditions for the generated parser

It is also possible to keep semantic information associated with the tokens currently on the parse stack in a user-defined C language *union*, if the members of the union are associated with the various names in the grammar file.

A declaration for a variable or constant uses the syntax of the C programming language:

TypeSpecifier Declarator ;

TypeSpecifier is a data type keyword and *Declarator* is the name of the variable or constant. Names can be any length and consist of letters, dots, underscores, and digits. A name cannot begin with a digit. Uppercase and lowercase letters are distinct.

Terminal (or token) names can be declared using the %**token** declaration and nonterminal names can be declared using the %**type** declaration. The %**type** declaration is not required for nonterminal names. Nonterminal names are defined automatically if they appear on the left side of at least one rule. Without declaring a name in the declarations section, you can use that name only as a nonterminal symbol. The #**include** statements are identical to C language syntax and perform the same function.

The **yacc** program has a set of keywords that define processing conditions for the generated parser. Each of the keywords begin with a % (percent sign) and is followed by a list of tokens or nonterminal names. These keywords are:

%**left**	Identifies tokens that are left-associative with other tokens.
%**nonassoc**	Identifies tokens that are not associative with other tokens.
%**right**	Identifies tokens that are right-associative with other tokens.
%**start**	Identifies a nonterminal name for the start symbol.
%**token**	Identifies the token names that the **yacc** command accepts. Declares all token names in the declarations section.
%**type**	Identifies the type of nonterminals. Type checking is performed when this construct is present.
%**union**	Identifies the yacc value stack as the union of the various type of values desired. By default, the values returned are integers. The effect of this construct is to provide the declaration of **YYSTYPE** directly from the input.

%{

Code

%} Copies the specified *Code* into the code file. This construct can be used to add C language declarations and definitions to the declarations section.

> **Note:** The %{ (percent sign, left bracket) and %} (percent sign, right bracket) symbols must appear on lines by themselves.

The %**token**, %**left**, %**right**, and %**nonassoc** keywords optionally support the name of a C union member (as defined by %**union**) called a <*Tag*> (literal angle brackets surrounding a union member name). The %**type** keyword requires a <*Tag*>. The use of <*Tag*> specifies that the tokens named on the line are to be of the same C type as the union member referenced by <*Tag*>. For example, the declaration:

```
%token [<Tag>] Name [Number] [Name [Number]]...
```

declares the *Name* parameter to be a token. If <*Tag*> is present, the C type for all tokens on this line are declared to be of the type referenced by <*Tag*>. If a positive integer, *Number*, follows the *Name* parameter, that value is assigned to the token.

All of the tokens on the same line have the same precedence level and associativity. The lines appear in the file in order of increasing precedence or binding strength. For example:

```
%left '+' '-'
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. The + (plus sign) and - (minus sign) are left associative and have lower precedence than * (asterisk) and / (slash), which are also left associative.

Defining Global Variables

To define variables to be used by some or all actions, as well as by the lexical analyzer, enclose the declarations for those variables between %{ (percent sign, left bracket) and %} (percent sign, right bracket) symbols. Declarations enclosed in these symbols are called global variables. For example, to make the **var** variable available to all parts of the complete program, use the following entry in the declarations section of the grammar file:

```
%{
int var = 0;
%}
```

Start Conditions

The parser recognizes a special symbol called the *start* symbol. The start symbol is the name of the rule in the rules section of the grammar file that describes the most general structure of the language to be parsed. Because it is the most general, this is the structure where the parser starts in its top-down analysis of the input stream. Declare the start symbol in the declarations section using the %**start** keyword. If you do not declare the name of the start symbol, the parser uses the name of the first grammar rule in the grammar file.

For example, when parsing a C language function, the most general structure for the parser to recognize is:

```
main()
{
        code_segment
}
```

The start symbol should point to the rule that describes this structure. All remaining rules in the file describe ways to identify lower-level structures within the function.

Token Numbers

Token numbers are nonnegative integers that represent the names of tokens. If the lexical analyzer passes the token number to the parser instead of the actual token name, both programs must agree on the numbers assigned to the tokens.

You can assign numbers to the tokens used in the **yacc** grammar file. If you do not assign numbers to the tokens, the **yacc** grammar file assigns numbers using the following rules:

1. A literal character is the numerical value of the character in the ASCII character set.
2. Other names are assigned token numbers starting at 257.

> **Note:** Do not assign a token number of 0. This number is assigned to the endmarker token. You cannot redefine it.

To assign a number to a token (including literals) in the declarations section of the grammar file, put a positive integer (not 0) immediately following the token name in the %token line. This integer is the token number of the name or literal. Each token number must be unique. All lexical analyzers used with the **yacc** command must return a 0 or a negative value for a token when they reach the end of their input.

yacc Rules

The rules section contains one or more grammar rules. Each rule describes a structure and gives it a name. A grammar rule has the form:

```
A : BODY;
```

where `A` is a nonterminal name, and `BODY` is a sequence of 0 or more names, literals, and semantic actions that can optionally be followed by precedence rules. Only the names and literals are required to form the grammar. Semantic actions and precedence rules are optional. The colon and the semicolon are required **yacc** punctuation.

Semantic actions allow you to associate actions to be performed each time a rule is recognized in the input process. An action can be an arbitrary C statement, and as such, perform input or output, call subprograms, or alter external variables. Actions can also refer to the actions of the parser; for example, shift and reduce.

Precedence rules are defined by the `%`**prec** keyword and change the precedence level associated with a particular grammar rule. The reserved symbol `%`**prec** can appear immediately after the body of the grammar rule and can be followed by a token name or a literal. The construct causes the precedence of the grammar rule to become that of the token name or literal.

Repeating Nonterminal Names

If several grammar rules have the same nonterminal name, use the | (pipe symbol) to avoid rewriting the left side. In addition, use the **;** (semicolon) only at the end of all rules joined by pipe symbols. For example, the grammar rules:

```
A  :  B  C  D  ;
A  :  E  F  ;
A  :  G  ;
```

can be given to the **yacc** command by using the pipe symbol as follows:

```
A  :  B  C  D
   |  E  F
   |  G
   ;
```

Using Recursion in a Grammar File

Recursion is the process of using a function to define itself. In language definitions, these rules normally take the form:

```
rule    :        EndCase
        |        rule EndCase
```

Therefore, the simplest case of the `rule` is the *EndCase*, but `rule` can also consist of more than one occurrence of *EndCase*. The entry in the second line that uses `rule` in the definition of `rule` is the recursion. The parser cycles through the input until the stream is reduced to the final *EndCase*.

When using recursion in a rule, always put the call to the name of the rule as the leftmost entry in the rule (as it is in the preceding example). If the call to the name of the rule occurs later in the line, such as in the following example, the parser may run out of internal stack space and stop.:

```
rule    :       EndCase
        |       EndCase rule
```

The following example defines the `line` rule as one or more combinations of a string followed by a newline character (`\n`):

```
lines   :       line
        |       lines line
        ;

line    :       string '\n'
        ;
```

Empty String

To indicate a nonterminal symbol that matches the empty string, use a **;** (semicolon) by itself in the body of the rule. To define a symbol `empty` that matches the `empty` string, use a rule similar to the following rule:

```
empty   :  ;
        | x;
```

OR

```
empty   :
        | x
        ;
```

End-of-Input Marker

When the lexical analyzer reaches the end of the input stream, it sends an end-of-input marker to the parser. This marker is a special token called *endmarker*, and has a token value of 0. When the parser receives an end-of-input marker, it checks to see that it has assigned all input to defined grammar rules and that the processed input forms a complete unit (as defined in the **yacc** grammar file). If the input is a complete unit, the parser stops. If the input is not a complete unit, the parser signals an error and stops.

The lexical analyzer must send the end-of-input marker at the appropriate time, such as the end of a file, or the end of a record.

yacc Actions

With each grammar rule, you can specify actions to be performed each time the parser recognizes the rule in the input stream. An action is a C language statement that does input and output, calls subprograms, and alters external vectors and variables. Actions return values and obtain the values returned by previous actions. The lexical analyzer can also return values for tokens.

Specify an action in the grammar file with one or more statements enclosed in `{}` (braces). The following examples are grammer rules with actions:

```
A  :  '('B')'
   {
     hello(1, "abc" );
   }
```

AND

```
XXX  :  YYY  ZZZ
     {
     printf("a message\n");
     flag = 25;
     }
```

Passing Values between Actions

To get values generated by other actions, an action can use the **yacc** parameter keywords that begin with a dollar sign (`$1`, `$2`, `...`). These keywords refer to the values returned by the components of the right side of a rule, reading from left to right. For example, if the rule is:

```
A  :  B  C  D  ;
```

then `$1` has the value returned by the rule that recognized B, `$2` has the value returned by the rule that recognized C, and `$3` the value returned by the rule that recognized D.

To return a value, the action sets the pseudo-variable $$ to some value. For example, the following action returns a value of 1:

```
{ $$ = 1;}
```

By default, the value of a rule is the value of the first element in it ($1). Therefore, you do not need to provide an action for rules that have the following form:

```
A : B ;
```

The following additional **yacc** parameter keywords beginning with a $ (dollar sign) allow for type checking:

- $<*Tag*>$
- $<*Tag*>*Number*

$<*Tag*>*Number* imposes on the reference the type of the union member referenced by <*Tag*>. This adds ***.tag*** to the reference so that the union member identified by *Tag* is accessed. This construct is equivalent to specifying $$**.***Tag* or $**1.***Tag*. You may use this construct when you use actions in the middle of rules where the return type cannot be specified through a %**type** declaration. If a %**type** has been declared for a nonterminal name, do not use the <*Tag*> construct; the union reference will be done automatically

Putting Actions in the Middle of Rules

To get control of the parsing process before a rule is completed, write an action in the middle of a rule. If this rule returns a value through the $ keywords, actions that follow this rule can use that value. This rule can also use values returned by actions that precede it. Therefore, the rule:

```
A  :  B
        {
           $$ =1;
        }
        C
   {
        x = $2;
        y = $3;
   }
   ;
```

sets x to 1 and y to the value returned by C. The value of rule A is the value returned by B, following the default rule.

Internally, the **yacc** command creates a new nonterminal symbol name for the action that occurs in the middle. It also creates a new rule matching this name to the empty string. Therefore, the **yacc** command treats the preceding program as if it were written in the following form:

```
$ACT  :  /* empty */
      {
         $$ = 1;
      }
      ;
A     :  B  $ACT  C
      {
         x = $2;
         y = $3;
      }
      ;
```

where $ACT is an empty action.

yacc Error Handling

When the parser reads an input stream, that input stream might not match the rules in the grammar file. The parser detects the problem as early as possible. If there is an error-handling subroutine in the grammar file, the parser can allow for entering the data again, skipping over the bad data, or initiating a

cleanup and recovery action. When the parser finds an error, for example, it may need to reclaim parse tree storage, delete or alter symbol table entries, and set switches to avoid generating further output.

When an error occurs, the parser stops unless you provide error-handling subroutines. To continue processing the input to find more errors, restart the parser at a point in the input stream where the parser can try to recognize more input. One way to restart the parser when an error occurs is to discard some of the tokens following the error. Then try to restart the parser at that point in the input stream.

The **yacc** command has a special token name, **error**, to use for error handling. Put this token in the rules file at places that an input error might occur so that you can provide a recovery subroutine. If an input error occurs in this position, the parser executes the action for the **error** token, rather than the normal action.

The following macros can be placed in **yacc** actions to assist in error handling:

YYERROR	Causes the parser to initiate error handling.
YYABORT	Causes the parser to return with a value of 1.
YYACCEPT	Causes the parser to return with a value of 0.
YYRECOVERING()	Returns a value of 1 if a syntax error has been detected and the parser has not yet fully recovered.

To prevent a single error from producing many error messages, the parser remains in error state until it processes 3 tokens following an error. If another error occurs while the parser is in the error state, the parser discards the input token and does not produce a message.

For example, a rule of the form:

```
stat  :  error ';'
```

tells the parser that when there is an error, it should skip over the token and all following tokens until it finds the next semicolon. All tokens after the error and before the next semicolon are discarded. After finding the semicolon, the parser reduces this rule and performs any cleanup action associated with it.

Providing for Error Correction

You can also allow the person entering the input stream in an interactive environment to correct any input errors by entering a line in the data stream again. For example:

```
input : error '\n'
        {
          printf(" Reenter last line: " );
         }
         input
       {
         $$ = $4;
       }
       ;
```

is one way to do this. However, in this example the parser stays in the error state for 3 input tokens following the error. If the corrected line contains an error in the first 3 tokens, the parser deletes the tokens and does not give a message. To allow for this condition, use the **yacc** statement:

```
yyerrok;
```

When the parser finds this statement, it leaves the error state and begins processing normally. The error-recovery example then becomes:

```
input : error '\n'
        {
          yyerrok;
          printf(" Reenter last line: " );
         }
```

```
  input
{
  $$ = $4;
}
;
```

Clearing the Look-Ahead Token

The *look-ahead token* is the next token that the parser examines. When an error occurs, the look-ahead token becomes the token at which the error was detected. However, if the error recovery action includes code to find the correct place to start processing again, that code must also change the look-ahead token. To clear the look-ahead token include the following statement in the error-recovery action:

```
yyclearin ;
```

Lexical Analysis for the yacc Command

You must provide a lexical analyzer to read the input stream and send tokens (with values, if required) to the parser that the **yacc** command generates. To build a lexical analyzer that works well with the parser that **yacc** generates, use the **lexlex** command.

The **lex** command generates a lexical analyzer called **yylex**. The **yylex**program must return an integer that represents the kind of token that was read. The integer is called the *token number*. In addition, if a value is associated with the token, the lexical analyzer must assign that value to the **yylval** external variable.

yacc-Generated Parser Operation

The **yacc** command turns the grammar file into a C language program. That program, when compiled and executed, parses the input according to the grammar specification given.

The parser is a finite state machine with a stack. The parser can read and remember the look-ahead token. The current state is always the state at the top of the stack. The states of the finite state machine are represented by small integers. Initially, the machine is in state 0, the stack contains only 0, and no look-ahead token has been read.

The machine can perform one of four actions:

shift *State*	The parser pushes the current state onto the stack, makes *State* the current state, and clears the look-ahead token.
reduce *Rule*	When the parser finds a string defined by *Rule* (a rule number) in the input stream, the parser replaces that string with *Rule* in the output stream.
accept	The parser looks at all input, matches it to the grammar specification, and recognizes the input as satisfying the highest-level structure (defined by the start symbol). This action appears only when the look-ahead token is the endmarker and indicates that the parser has successfully done its job.
error	The parser cannot continue processing the input stream and still successfully match it with any rule defined in the grammar specification. The input tokens the parser looked at, together with the look-ahead token, cannot be followed by anything that would result in valid input. The parser reports an error and attempts to recover the situation and resume parsing.

The parser performs the following actions during one process step:

1. Based on its current state, the parser decides whether it needs a look-ahead token to decide the action to be taken. If the parser needs a look-ahead token and does not have one, it calls the **yylex** subroutine to obtain the next token.
2. Using the current state, and the look-ahead token if needed, the parser decides on its next action and carries it out. As a result, states may be pushed onto or popped off the stack, and the look-ahead token may be processed or left alone.

Shift

The **shift** action is the most common action the parser takes. Whenever the parser does a shift, there is always a look-ahead token. For example, consider the following grammar specification rule:

```
IF shift 34
```

If the parser is in the state that contains this rule and the look-ahead token is `IF`, the parser:

1. Pushes the current state down on the stack.
2. Makes state `34` the current state (puts it at the top of the stack).
3. Clears the look-ahead token.

Reduce

The **reduce** action keeps the stack from growing too large. The parser uses reduce actions after matching the right side of a rule with the input stream. The parser is then ready to replace the characters in the input stream with the left side of the rule. The parser may have to use the look-ahead token to decide if the pattern is a complete match.

Reduce actions are associated with individual grammar rules. Because grammar rules also have small integer numbers, you can easily confuse the meanings of the numbers in the two actions, **shift** and **reduce**. For example, the following action refers to grammar rule `18`:

```
. reduce 18
```

The following action refers to state `34`:

```
IF shift 34
```

For example, to reduce the following rule, the parser pops off the top three states from the stack:

```
A  :  x y z ;
```

The number of states popped equals the number of symbols on the right side of the rule. These states are the ones put on the stack while recognizing `x`, `y`, and `z`. After popping these states, a state is uncovered, which is the state the parser was in before beginning to process the rule, that is, the state that needed to recognize rule `A` to satisfy its rule. Using this uncovered state and the symbol on the left side of the rule, the parser performs an action called **goto**, which is similar to a shift of `A`. A new state is obtained, pushed onto the stack, and parsing continues.

The **goto** action is different from an ordinary shift of a token. The look-ahead token is cleared by a shift but is not affected by a **goto** action. When the three states are popped, the uncovered state contains an entry such as:

```
A  goto 20
```

This entry causes state `20` to be pushed onto the stack and become the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the parser executes the code that you included in the rule before adjusting the stack. Another stack running in parallel with the stack holding the states holds the values returned from the lexical analyzer and the actions. When a shift takes place, the **yylval** external variable is copied onto the stack holding the values. After executing the code that you provide, the parser performs the reduction. When the parser performs the **goto** action, it copies the **yylval** external variable onto the value stack. The **yacc** keywords that begin with `$` refer to the value stack.

Using Ambiguous Rules in the yacc Program

A set of grammar rules is *ambiguous* if any possible input string can be structured in two or more different ways. For example, the grammar rule:

```
expr : expr '-' expr
```

states a rule that forms an arithmetic expression by putting two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not specify how to structure all complex inputs. For example, if the input is:

```
expr - expr - expr
```

a program could structure this input as either left associative:

```
( expr - expr ) - expr
```

or as right associative:

```
expr - ( expr - expr )
```

and produce different results.

Parser Conflicts

When the parser tries to handle an ambiguous rule, confusion occurs over which of its four actions to perform when processing the input. Two major types of conflict develop:

shift/reduce conflict	A rule can be evaluated correctly using either a **shift** action or a **reduce** action, but the result is different.
reduce/reduce conflict	A rule can be evaluated correctly using one of two different reduce actions, producing two different actions.

A **shift/shift** conflict is not possible. The **shift/reduce** and **reduce/reduce** conflicts result from a rule that is not completely stated. For example, using the ambiguous rule stated in the previous section, if the parser receives the input:

```
expr - expr - expr
```

after reading the first three parts the parser has:

```
expr - expr
```

which matches the right side of the preceding grammar rule. The parser can reduce the input by applying this rule. After applying the rule, the input becomes:

```
expr
```

which is the left side of the rule. The parser then reads the final part of the input:

```
- expr
```

and reduces it. This produces a left associative interpretation.

However, the parser can also look ahead in the input stream. If, when the parser receives the first three parts:

```
expr - expr
```

it reads the input stream until it has the next two parts, it then has the following input:

```
expr - expr - expr
```

Applying the rule to the rightmost three parts reduces them to `expr`. The parser then has the expression:

```
expr - expr
```

Reducing the expression once more produces a right associative interpretation.

Therefore, at the point where the parser has read only the first three parts, it can take two valid actions: a **shift** or a **reduce**. If the parser has no rule to decide between the two actions, a **shift/reduce** conflict results.

A similar situation occurs if the parser can choose between two valid reduce actions. That situation is called a **reduce/reduce** conflict.

How the Parser Responds to Conflicts

When **shift/reduce** or **reduce/reduce** conflicts occur, the **yacc** command produces a parser by selecting one of the valid steps wherever it has a choice. If you do not provide a rule that makes the choice, **yacc** uses two rules:

1. In a **shift/reduce** conflict, choose the shift.
2. In a **reduce/reduce** conflict, reduce by the grammar rule that can be applied at the earliest point in the input stream.

Using actions within rules can cause conflicts if the action must be performed before the parser is sure which rule is being recognized. In such cases, the preceding rules result in an incorrect parser. For this reason, the **yacc** program reports the number of **shift/reduce** and **reduce/reduce** conflicts resolved using the preceding rules.

Turning on Debug Mode for a yacc-Generated Parser

You can access the debugging code either by invoking the **yacc** command with the **-t** option or compiling the **y.tab.c** file with **-DYYDEBUG**.

For normal operation, the **yydebug** external integer variable is set to 0. However, if you set it to a nonzero value, the parser generates a description of:

- The input tokens it receives.
- The actions it takes for each token while parsing an input stream.

Set this variable in one of the following ways:

- Put the following C language statement in the declarations section of the **yacc** grammar file:
  ```
  int yydebug = 1;
  ```
- Use **dbx** to execute the final parser, and set the variable ON or OFF using **dbx** commands.

Example Program for the lex and yacc Programs

This section describes example programs for the **lex** and **yacc** commands. Together, these example programs create a simple, desk-calculator program that performs addition, subtraction, multiplication, and division operations. This calculator program also allows you to assign values to variables (each designated by a single, lowercase letter) and then use the variables in calculations. The files that contain the example **lex** and **yacc** programs are:

File	Content
calc.lex (“Lexical Analyzer Source Code” on page 299)	Specifies the **lex** command specification file that defines the lexical analysis rules.
calc.yacc (“Parser Source Code” on page 297)	Specifies the **yacc** command grammar file that defines the parsing rules, and calls the **yylex** subroutine created by the **lex** command to provide input.

The following descriptions assume that the **calc.lex** and **calc.yacc** example programs are found in your current directory.

Compiling the Example Program

Perform the following steps, in order, to create the desk calculator example program:

1. Process the **yacc** grammar file using the **-d** optional flag (which tells the **yacc** command to create a file that defines the tokens used in addition to the C language source code):

   ```
   yacc -d calc.yacc
   ```

2. Use the **ls** command to verify that the following files were created:

y.tab.c	The C language source file that the **yacc** command created for the parser.
y.tab.h	A header file containing define statements for the tokens used by the parser.

3. Process the **lex** specification file:

   ```
   lex calc.lex
   ```

4. Use the **ls** command to verify that the following file was created:

lex.yy.c	The C language source file that the **lex** command created for the lexical analyzer.

5. Compile and link the two C language source files:

   ```
   cc y.tab.c lex.yy.c
   ```

6. Use the **ls** command to verify that the following files were created:

y.tab.o	The object file for the **y.tab.c** source file
lex.yy.o	The object file for the **lex.yy.c** source file
a.out	The executable program file

To then run the program directly from the **a.out** file, enter:

```
$ a.out
```

Or, to move the program to a file with a more descriptive name, as in the following example, and run it, enter:

```
$ mv a.out calculate
$ calculate
```

In either case, after you start the program, the cursor moves to the line below the $ (command prompt). Then enter numbers and operators in calculator fashion. When you press the Enter key, the program displays the result of the operation. After you assign a value to a variable:

```
m=4 <enter>
_
```

the cursor moves to the next line. When you use the variable in subsequent calculations, it will have the assigned value:

```
m+5 <enter>
9
_
```

Parser Source Code

The following example shows the contents of the **calc.yacc** file. This file has entries in all three sections of a **yacc** grammar file: declarations, rules, and programs.

```
%{
#include <stdio.h>
int regs[26];
int base;
%}
%start list
%token DIGIT LETTER
```

```
%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS  /*supplies precedence for unary minus */
%%                   /* beginning of rules section */
list:                        /*empty */
        |
        list stat '\n'
        |
        list error '\n'
        {
          yyerrok;
        }
        ;
stat:   expr
        {
          printf("%d\n",$1);
        }
        |
        LETTER '=' expr
        {
          regs[$1] = $3;
        }
        ;
expr:   '(' expr ')'
        {
          $$ = $2;
        }
        |
        expr '*' expr
        {
          $$ = $1 * $3;
        }
        |
        expr '/' expr
        {
          $$ = $1 / $3;
        }
        |
        expr '%' expr
        {
          $$ = $1 % $3;
        }
        |
        expr '+' expr
        {
          $$ = $1 + $3;
        }
         |
        expr '-' expr
        {
          $$ = $1 - $3;
        }
        |
        expr '&' expr
        {
          $$ = $1 & $3;
        }
        |
        expr '|' expr
        {
          $$ = $1 | $3;
        }
        |
```

```
        '-' expr %prec UMINUS
         {
           $$ = -$2;
         }
         |
         LETTER
         {
           $$ = regs[$1];
         }
         |
         number
         ;
number:  DIGIT
         {
           $$ = $1;
           base = ($1==0) ? 8 : 10;
         }       |
         number DIGIT
         {
           $$ = base * $1 + $2;
         }
         ;
%%
main()
{
 return(yyparse());
}
yyerror(s)
char *s;
{
  fprintf(stderr, "%s\n",s);
}
yywrap()
{
  return(1);
}
```

Declarations Section: This section contains entries that:

- Include standard I/O header file.
- Define global variables.
- Define the `list` rule as the place to start processing.
- Define the tokens used by the parser.
- Define the operators and their precedence.

Rules Section: The rules section defines the rules that parse the input stream.

Programs Section: The programs section contains the following subroutines. Because these subroutines are included in this file, you do not need to use the **yacc** library when processing this file.

main	The required main program that calls the **yyparse** subroutine to start the program.
yyerror(s)	This error-handling subroutine only prints a syntax error message.
yywrap	The wrap-up subroutine that returns a value of 1 when the end of input occurs.

Lexical Analyzer Source Code

Following are the contents of the **calc.lex** file. This file contains include statements for standard input and output, as well as for the **y.tab.h** file. The **yacc** program generates that file from the **yacc** grammar file information if you use the **-d** flag with the **yacc** command. The **y.tab.h** file contains definitions for the tokens that the parser program uses. In addition, **calc.lex** contains the rules to generate these tokens from the input stream.

```
%{

#include <stdio.h>
#include "y.tab.h"
int c;
extern int yylval;
%}
%%
" "       ;
[a-z]     {
            c = yytext[0];
            yylval = c - 'a';
            return(LETTER);
          }
[0-9]     {
            c = yytext[0];
            yylval = c - '0';
            return(DIGIT);
          }
[^a-z0-9\b]    {
                 c = yytext[0];
                 return(c);
               }
```

Chapter 13. Logical Volume Programming

This chapter provides an introduction to programming considerations for the Logical Volume Manager (LVM), which consists of the library of LVM subroutines and the logical volume device driver.

The Logical Volume Manager (LVM) consists of two main components.

The first is the library of LVM subroutines. These subroutines define volume groups and maintain the logical and physical volumes of volume groups. They are used by the system management commands to perform system management tasks for the logical and physical volumes of a system. The programming interface for the library of LVM subroutines is available to anyone who wishes to provide alternatives or to expand the function of the system management commands for logical volumes.

The other main component of LVM is the logical volume device driver. The logical volume device driver is a pseudo-device driver that processes all logical I/O. It exists as a layer between the file system and the disk device drivers. The logical volume device driver converts a logical address to a physical address, handles mirroring and bad-block relocation, and then sends the I/O request to the specific disk device driver. Interfaces to the logical volume device driver are provided by the **open**, **close**, **read**, **write**, and **ioctl** subroutines.

A description of the **readx** and **writex** extension parameters and those **ioctl** operations specific to the logical volume device driver is found in Understanding the Logical Volume Device Driver in *AIX 5L Version 5.1 Kernel Extensions and Device Support Programming Concepts*.

See the Logical Volume Storage Overview in *AIX 5L Version 5.1 System Management Guide: Operating System and Devices* for more information on logical volumes.

List of Logical Volume Subroutines

The library of LVM subroutines is a main component of the Logical Volume Manager.

LVM subroutines define and maintain the logical and physical volumes of a volume group. They are used by the system management commands to perform system management for the logical and physical volumes of a system. The programming interface for the library of LVM subroutines is available to anyone who wishes to provide alternatives to or expand the function of the system management commands for logical volumes.

Note: The LVM subroutines use the **sysconfig** system call, which requires root user authority, to query and update kernel data structures describing a volume group. You must have root user authority to use the services of the LVM subroutine library.

The following services are available:

lvm_querylv	Queries a logical volume and returns all pertinent information.
lvm_querypv	Queries a physical volume and returns all pertinent information.
lvm_queryvg	Queries a volume group and returns pertinent information.
lvm_queryvgs	Queries the volume groups of the system and returns information for groups that are varied on-line.

Chapter 14. make Command

This chapter provides information about simplifying the recompiling and relinking processes using the **make** command. It is a useful utility that can save you time when managing projects.

The **make** program is most useful for medium-sized programming projects. It does not solve the problems of maintaining more than one source version and of describing large programs (see **sccs** command).

The **make** command assists you in maintaining a set of programs, usually pertaining to a particular software project. It does this by building up-to-date versions of programs.

In any project, you normally link programs from object files and libraries. Then, after modifying a source file, you recompile some of the sources and relink the program as often as required.

The **make** command simplifies the process of recompiling and relinking programs. It allows you to record, once only, specific relationships among files. You can then use the **make** command to automatically perform all updating tasks.

Using the **make** command to maintain programs, you can:

- Combine instructions for creating a large program in a single file.
- Define macros to use within the **make** command description file.
- Use shell commands to define the method of file creation, or use the **make** program to create many of the basic types of files.
- Create libraries.

The **make** command requires a description file, file names, specified rules to tell the **make** program how to build many standard types of files, and time stamps of all system files.

Creating a Description File

The **make** program uses information from a description file that you create to build a file containing the completed program, which is then called a *target* file. The description file tells the **make** command how to build the target file, which files are involved, and what their relationships are to the other files in the procedure. The description file contains the following information:

- Target file name
- Parent file names that make up the target file
- Commands that create the target file from the parent files
- Definitions of macros in the description file
- User-specified rules for building target files

By checking the dates of the parent files, the **make** program determines which files to create to get an up-to-date copy of the target file. If any parent file was changed more recently than the target file, the **make** command creates the files affected by the change, including the target file.

If you name the description file **makefile** or **Makefile** and are working in the directory containing that description file, enter:

```
make
```

to update the first target file and its parent files. Updating occurs regardless of the number of files changed since the last time the **make** command created the target file. In most cases, the description file is easy to write and does not change often.

To keep many different description files in the same directory, name them differently. Then, enter:

```
make -f Desc-File
```

substituting the name of the description file for the *Desc-File* variable.

Format of a make Description File Entry

The general form of an entry is:

```
target1 [target2..]:[:] [parent1..][; command]...
[(tab) commands]
```

Items inside brackets are optional. Targets and parents are file names (strings of letters, numbers, periods, and slashes). The **make** command recognizes wildcard characters such as * (asterisk) and ? (question mark). Each line in the description file that contains a target file name is called a *dependency line*. Lines that contain commands must begin with a tab character.

> **Note:** The **make** command uses the $ (dollar sign) to designate a macro. Do not use that symbol in file names of target or parent files, or in commands in the description file unless you are using a predefined **make** command macro.

Begin comments in the description file with a # (pound sign). The **make** program ignores the # and all characters that follow it. The **make** program also ignores blank lines.

Except for comment lines, you can enter lines longer than the line width of the input device. To continue a line on the next line, put a \ (backslash) at the end of the line to be continued.

Using Commands in a make Description File

A command is any string of characters except a # (pound sign) or a new-line character. A command can use a # only if it is in quotes. Commands can appear either after a semicolon on a dependency line or on lines beginning with a tab that immediately follows a dependency line.

When defining the command sequence for a particular target, specify one command sequence for each target in the description file, or else separate command sequences for special sets of dependencies. Do not do both.

To use one command sequence for every use of the target file, use a single : (colon) following the target name on the dependency line. For example:

```
test:        dependency list1...
         command list...
            .
            .
            .
test:        dependency list2...
```

defines a target name, `test`, with a set of parent files and a set of commands to create the file. The target name, `test`, can appear in other places in the description file with another dependency list.

However, that name cannot have another command list in the description file. When one of the files that `test` depends on changes, the **make** command runs the commands in that one command list to create the target file named `test`.

To specify more than one set of commands to create a particular target file, enter more than one dependency definition. Each dependency line must have the target name, followed by :: (two colons), a dependency list, and a command list that the **make** command uses if any of the files in the dependency list changes. For example:

```
test::      dependency list1...
         command list1...
test::      dependency list2...
         command list2...
```

defines two separate processes to create the target file, `test`. If any of the files in `dependency list1` changes, the **make** command runs `command list1`. If any of the files in `dependency list2` changes, the **make** command runs `command list2`. To avoid conflicts, a parent file cannot appear in both `dependency list1` and `dependency list2`.

> **Note:** The **make** command passes the commands from each command line to a new shell. Be careful when using commands that have meaning only within a single shell process; for example, **cd** and shell commands. The **make** program forgets these results before running the commands on the next line.

To group commands together, use the \ (backslash) at the end of a command line. The **make** program then continues that command line into the next line in the description file. The shell sends both of these lines to a single new shell.

Calling the make Program from a Description File

To nest calls to the **make** program within a **make** command description file, include the **$(MAKE)** macro in one of the command lines in the file.

If the **-n** flag is set when the **$(MAKE)** macro is found, the new copy of the **make** command does not execute any of its commands, except another **$(MAKE)** macro. Use this characteristic to test a set of description files that describe a program. Enter the command:

```
make -n
```

The **make** program does not perform any of the program operations. However, it does write all of the steps needed to build the program, including output from lower-level calls to the **make** command.

Preventing the make Program from Writing Commands

To prevent the **make** program from writing the commands while it runs, do any of the following:

- Use the **-s** flag on the command line when using the **make** command.
- Put the fake target name **.SILENT** on a dependency line by itself in the description file. Because **.SILENT** is not a real target file, it is called a fake target. If **.SILENT** has prerequisites, the **make** command does not display any of the commands associated with them.
- Put an @ (at sign) in the first character position of each line in the description file that the **make** command should not write.

Preventing the make Program from Stopping on Errors

The **make** program normally stops if any program returns a nonzero error code. Some programs return status that has no meaning.

To prevent the **make** command from stopping on errors, do any of the following:

- Use the **-i** flag with the **make** command on the command line.
- Put the fake target name **.IGNORE** on a dependency line by itself in the description file. Because **.IGNORE** is not a real target file, it is called a fake target. If **.IGNORE** has prerequisites, the **make** command ignores errors associated with them.
- Put a - (minus sign) in the first character position of each line in the description file where the **make** command should not stop on errors.

Example of a Description File

For example, a program named **prog** is made by compiling and linking three C language files `x.c`, `y.c`, and `z.c`. The `x.c` and `y.c` files share some declarations in a file named `defs`. The `z.c` file does not share those declarations. The following is an example of a description file, which creates the **prog** program:

```
# Make prog from 3 object files
prog: x.o y.o z.o
# Use the cc program to make prog
    cc  x.o y.o z.o -o prog
# Make x.o from 2 other files
x.o:   x.c defs
# Use the cc program to make x.o
    cc -c x.c
# Make y.o from 2 other files
y.o: y.c defs
# Use the cc program to make y.o
    cc  -c y.c
# Make z.o from z.c
z.o:   z.c
# Use the cc program to make z.o
    cc  -c z.c
```

If this file is called `makefile`, just enter the command:

```
make
```

to update the **prog** program after making changes to any of the four source files: `x.c`, `y.c`, `z.c`, or `defs`.

Making the Description File Simpler

To make this file simpler, use the internal rules of the **make** program. Based on file-system naming conventions, the **make** command recognizes three **.c** files corresponding to the needed **.o** files. This command can also generate an object from a source file, by issuing a **cc -c** command.

Based on these internal rules, the description file becomes:

```
# Make prog from 3 object files
prog:  x.o y.o z.o
# Use the cc program to make prog
    cc  x.o y.o z.o -o prog
# Use the file defs and the .c file
# when making x.o and y.o
x.o y.o:   defs
```

Internal Rules for the make Program

The internal rules for the **make** program are in a file that looks like a description file. When the **-r** flag is specified, the **make** program does not use the internal rules file. You must supply the rules to create the files in your description file. The internal-rules file contains a list of file-name suffixes (such as **.o**, or **.a**) that the **make** command understands, plus rules that tell the **make** command how to create a file with one suffix from a file with another suffix. If you do not change the list, the **make** command understands the following suffixes:

.a Archive library.
.C C++ source file.
.C\~ SCCS file containing C++ source file.
.c C source file.
.c~ Source Code Control System (SCCS) file containing C source file.
.f FORTRAN source file.
.f~ SCCS file containing FORTRAN source file.
.h C language header file.

.h~ SCCS file containing C language header file.
.l **lex** source grammar.
.l~ SCCS file containing **lex** source grammar.
.o Object file.
.s Assembler source file.
.s~ SCCS file containing assembler source file.
.sh Shell-command source file.
.sh~ SCCS file containing shell-command source file.
.y **yacc-c** source grammar.
.y~ SCCS file containing **yacc-c** source grammar.

The list of suffixes is similar to a dependency list in a description file, and follows the fake target name **.SUFFIXES**. Because the **make** command looks at the suffixes list in left-to-right order, the order of the entries is important.

The **make** program uses the first entry in the list that satisfies the following two requirements:

- The entry matches input and output suffix requirements for the current target and dependency files.
- The entry has a rule assigned to it.

The **make** program creates the name of the rule from the two suffixes of the files that the rule defines. For example, the name of the rule to transform a **.c** file to an **.o** file is **.c.o**.

To add more suffixes to the list, add an entry for the fake target name **.SUFFIXES** in the description file. For a **.SUFFIXES** line without any suffixes following the target name in the description file, the **make** command erases the current list. To change the order of the names in the list, erase the current list and then assign a new set of values to **.SUFFIXES**.

Example of Default Rules File

The following example shows a portion of the default rules file:

```
# Define suffixes that make knows.
.SUFFIXES:  .o .C .C\~ .c .c~ .f .f~ .y .y~ .l .l~ .s .s~ .sh .sh~ .h .h~ .a
 #Begin macro definitions for
#internal macros
YACC=yacc
YFLAGS=
ASFLAGS=
LEX=lex
LFLAGS=
CC=cc
CCC=xlC
AS=as
CFLAGS=
CCFLAGS=
# End macro definitions for
# internal macros
# Create a .o file from a .c
# file with the cc program.
c.o:
         $(CC) $(CFLAGS) -c $<
# Create a .o file from
# a .s file with the assembler.
s.o:
         $(AS)$(ASFLAGS) -o $@ $<
.y.o:
# Use yacc to create an intermediate file
         $(YACC) $(YFLAGS) $<
# Use cc compiler
         $(CC) $(CFLAGS) -c y.tab.c
```

```
# Erase the intermediate file
         rm y.tab.c
# Move to target file
         mv y.tab.o $@.
.y.c:
# Use yacc to create an intermediate file
         $(YACC) $(YFLAGS) $<
# Move to target file
         mv y.tab.c $@
```

Single-Suffix Rules

The **make** program has a set of single-suffix rules to build source files directly into a target file name that does not have a suffix (command files, for example). The **make** program also has rules to change the following source files with suffix to object files without a suffix:

.C: From a C++ language source file.
.C\~: From an SCCS C++ language source file.
.c: From a C language source file.
.c~: From an SCCS C language source file.
.sh: From a shell file.
.sh~: From an SCCS shell file.

For example, to maintain the `cat` program, enter:

```
make cat
```

if all of the needed source files are in the current directory.

Using the Make Command with Archive Libraries

Use the **make** command to build libraries and library files. The **make** program recognizes the suffix **.a** as a library file. The internal rules for changing source files to library files are:

.C.a C++ source to archive.
.C\~.a SCCS C++ source to archive.
.c.a C source to archive.
.c~.a SCCS C source to archive.
.s~.a SCCS assembler source to archive.
.f.a Fortran source to archive.
.f~.a SCCS Fortran source to archive.

Changing Macros in the Rules File

The **make** program uses macro definitions in the rules file. To change these macro definitions, enter new definitions for each macro on the command line or in the description file. The **make** program uses the following macro names to represent the language processors that it uses:

AS For the assembler.
CC For the C compiler.
CCC For the C++ compiler.
YACC For the **yacc** command.
LEX For the **lex** command.

The **make** program uses the following macro names to represent the flags that it uses:

CFLAGS For C compiler flags.
CCFLAGS For C++ compiler flags.

YFLAGS For **yacc** command flags.
LFLAGS For **lex** command flags.

Therefore, the command:

```
make "CC=NEWCC"
```

directs the **make** command to use the `NEWCC` program in place of the usual C language compiler. Similarly, the command:

```
make "CFLAGS=-O"
```

directs the **make** command to optimize the final object code produced by the C language compiler.

To review the internal rules that the **make** command uses, refer to the **/usr/ccs/lib/make.cfg** file.

Defining Default Conditions in a Description File

When the **make** command creates a target file but cannot find commands in the description file or internal rules to create a file, it looks at the description file for default conditions. To define the commands that the **make** command performs in this case, use the **.DEFAULT** target name in the description file:

```
.DEFAULT:
                command
                command
                   .
                   .
                   .
```

Because **.DEFAULT** is not a real target file, it is called a *fake target*. Use the **.DEFAULT** fake target name for an error-recovery routine or for a general procedure to create all files in the program that are not defined by an internal rule of the **make** program.

Including Other Files in a Description File

Include files other than the current description file by using the word **include** as the first word on any line in the description file. Follow the word with a blank or a tab, and then the file name for the **make** command to include in the operation.

Note: Only one file is supported per **include** statement.

For example:

```
include /home/tom/temp
include /home/tom/sample
```

directs the **make** command to read the `temp` and `sample` files and the current description file to build the target file.

Do not use more than 16 levels of nesting with the include files feature.

Defining and Using Macros in a Description File

A *macro* is a name (or label) to use in place of several other names. It is a way of writing the longer string of characters in shorthand. To define a macro:

1. Start a new line with the name of the macro.
2. Follow the name with an = (equal sign).
3. To the right of the = (equal sign), enter the string of characters that the macro name represents.

The macro definition can contain blanks before and after the = (equal sign) without affecting the result. The macro definition cannot contain a : (colon) or a tab before the = (equal sign).

The following are examples of macro definitions:

```
# Macro -"2" has a value of "xyz"
2 = xyz

# Macro "abc" has a value of "-ll -ly"
abc = -ll -ly

# Macro "LIBES" has a null value
LIBES =
```

A macro that is named, but not defined, has the same value as the null string.

Using Macros in a Description File

After defining a macro in a description file, use the macro in description file commands by putting a $ (dollar sign) before the name of the macro. If the macro name is longer than one character, put ( ) (parentheses) or { } (braces) around it. The following are examples of using macros:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two examples in the previous list have the same effect.

The following fragment shows how to define and use some macros:

```
# OBJECTS is the 3 files x.o, y.o and
# z.o (previously compiled)
OBJECTS = x.o y.o z.o
# LIBES is the standard library
LIBES = -lc
# prog depends on x.o y.o and z.o
prog:  $(OBJECTS)
# Link and load the 3 files with
# the standard library to make prog
      cc $(OBJECTS) $(LIBES) -o prog
```

The **make** program using this description file links and loads the three object files (`x.o`, `y.o`, and `z.o`) with the **libc.a** library.

A macro definition entered on the command line overrides any duplicate macro definitions in the description file. Therefore, the command:

```
make "LIBES= -ll"
```

loads the files with the **lex** (`-ll`) library.

> **Note:** When entering macros with blanks in them on the command line, put ″ ″ (double quotation marks) around the macro. Without the double quotation marks, the shell interprets the blanks as parameter separators and not as part of the macro.

The **make** command handles up to 10 levels of nested macro expansion. Based on the definitions in the following example:

```
macro1=value1
macro2=macro1
```

the expression `$($(macro2))` would evaluate to `value1`.

The evaluation of a macro occurs each time the macro is referenced. It is not evaluated when it is defined. If a macro is defined but never used, it will never be evaluated. This is especially important if the macro is assigned values that will be interpreted by the shell, particularly if the value might change. A variable declaration such as:

```
OBJS = 'ls *.o'
```

could change in value if referenced at different times during the process of building or removing object files. It does not hold the value of the **ls** command at the time the **OBJS** macro is defined.

Internal Macros

The **make** program has built-in macro definitions for use in the description file. These macros help specify variables in the description file. The **make** program replaces the macros with one of the following values:

$@	Name of the current target file.
$$@	Label name on the dependency line.
$?	Names of the files that have changed more recently than the target.
$<	Parent file name of the out-of-date file that caused a target file to be created.
$*	Name of the current parent file without the suffix.
$%	Name of an archive library member.

Target File Name

If the **$@** macro is in the command sequence in the description file, the **make** command replaces the symbol with the full name of the current target file before passing the command to the shell to be run. The **make** program replaces the symbol only when it runs commands from the description file to create the target file.

Label Name

If the **$$@** macro is on the dependency line in a description file, the **make** command replaces this symbol with the label name that is on the left side of the colon in the dependency line. For example, if the following is included on a dependency line:

```
cat:     $$@.c
```

the **make** program translates it to:

```
cat:     cat.c
```

when the **make** command evaluates the expression. Use this macro to build a group of files, each of which has only one source file. For example, to maintain a directory of system commands, use a description file like:

```
# Define macro CMDS as a series
# of command names
CMDS = cat dd echo date cc cmp comm ar ld chown
# Each command depends on a .c file
$(CMDS):      $$@.c
# Create the new command set by compiling the out of
# date files ($?) to the target file name ($@)
         $(CC) -O $? -o $@
```

The **make** program changes the **$$(@F)** macro to the file part of **$@** when it runs. For example, use this symbol when maintaining the **usr/include** directory while using a description file in another directory. That description file is similar to the following:

```
# Define directory name macro INCDIR
INCDIR = /usr/include
# Define a group of files in the directory
# with the macro name INCLUDES
INCLUDES = \
         $(INCDIR)/stdio.h \
         $(INCDIR)/pwd.h \
```

```
        $(INCDIR)/dir.h \
        $(INCDIR)/a.out.h \
# Each file in the list depends on a file
# of the same name in the current directory
$(INCLUDES):       $$(@F)
# Copy the younger files from the current
# directory to /usr/include
        cp $? $@
# Set the target files to read only status
        chmod 0444 $@
```

This description file creates a file in the **/usr/include** directory when the corresponding file in the current directory has been changed.

Younger Files

If the **$?** macro is in the command sequence in the description file, the **make** command replaces the symbol with a list of parent files that have been changed since the target file was last changed. The **make** program replaces the symbol only when it runs commands from the description file to create the target file.

First Out-of-Date File

If the **$<** macro is in the command sequence in the description file, the **make** command replaces the symbol with the name of the file that started the file creation. The file name is the name of the parent file that was out-of-date with the target file, and therefore caused the **make** command to create the target file again.

In addition, use a letter (**D** or **F**) after the < (less-than sign) to get either the directory name (**D**) or the file name (**F**) of the first out-of-date file. For example, if the first out-of-date file is:

```
/home/linda/sample.c
```

then the **make** command gives the following values:

```
$(<D)   =   /home/linda
$(<F)   =   sample.c
$<      =   /home/linda/sample.c
```

The **make** program replaces this symbol only when the program runs commands from its internal rules or from the **.DEFAULT** list.

Current File-Name Prefix

If the **$*** macro is in the command sequence in the description file, the **make** command replaces the symbol with the file-name part (without the suffix) of the parent file that the **make** command is currently using to generate the target file. For example, if the **make** command is using the file:

```
test.c
```

then the **$*** macro represents the file name `test`.

In addition, use a letter (**D** or **F**) after the * (asterisk) to get either the directory name (**D**) or the file name (**F**) of the current file.

For example, the **make** command uses many files (specified either in the description file or in the internal rules) to create a target file. Only one of those files (the current file) is used at any moment. If that current file is:

```
/home/tom/sample.c
```

then the **make** command gives the following values for the macros:

```
$(*D)  =  /home/tom
$(*F)  =  sample
$*     =  /home/tom/sample
```

The **make** program replaces this symbol only when running commands from its internal rules (or from the **.DEFAULT** list), but not when running commands from a description file.

Archive Library Member

If the **$%** macro is in a description file, and the target file is an archive library member, the **make** command replaces the macro symbol with the name of the library member. For example, if the target file is:

```
lib(file.o)
```

then the **make** command replaces the **$%** macro with the member name, `file.o`.

Changing Macro Definitions in a Command

When macros in the shell commands are defined in the description file, you can change the values that the **make** command assigns to the macro. To change the assignment of the macro, put a : (colon) after the macro name, followed by a replacement string. The form is as follows:

```
$(macro:string1=string2)
```

When the **make** command reads the macro and begins to assign the values to the macro based on the macro definition, the command replaces each `string1` in the macro definition with a value of `string2`. For example, if the description file contains the macro definition:

```
FILES=test.o sample.o form.o defs
```

you can replace the `form.o` file with a new file, `input.o`, by using the macro in the description-file commands, as follows:

```
cc -o $(FILES:form.o=input.o)
```

Changing the value of a macro in this manner is useful when maintaining archive libraries. For more information, see the **ar** command.

How the make Command Creates a Target File

The **make** command creates a file containing the completed program called a *target* file, using a step-by-step procedure.

The **make** program:

1. Finds the name of the target file in the description file or in the **make** command
2. Ensures that the files on which the target file depends exist and are up-to-date
3. Determines if the target file is up-to-date with the files it depends on.

If the target file or one of the parent files is out-of-date, the **make** program creates the target file using one of the following:

- Commands from the description file
- Internal rules to create the file (if they apply)
- Default rules from the description file.

If all files in the procedure are up-to-date when running the **make** program, the **make** command displays a message to indicate that the file is up-to-date, and then stops. If some files have changed, the **make** command builds only those files that are out-of-date. The command does not rebuild files that are already current.

When the **make** program runs commands to create a target file, it replaces macros with their values, writes each command line, and then passes the command to a new copy of the shell.

Using the make Command with Source Code Control System (SCCS) Files

The SCCS command and file system is primarily used to control access to a file, track who altered the file, why it was altered, and what was altered. An SCCS file is any text file controlled with SCCS commands. Using non-SCCS commands to edit SCCS files can damage the SCCS files. See “Chapter 23. Source Code Control System (SCCS)” on page 629 to learn more about SCCS.

All SCCS files use the prefix **s.** to set them apart from regular text files. The **make** program does not recognize references to prefixes of file names. Therefore, do not refer to SCCS files directly within the **make** command description file. The **make** program uses a different suffix, the ~ (tilde), to represent SCCS files. Therefore, **.c~.o** refers to the rule that transforms an SCCS C language source file into an object file. The internal rule is:

```
.c~.o:
        $(GET) $(GFLAGS) -p  $<  >$*.c
        $(CC) $(CFLAGS) -c $*.c
        -rm -f $*.c
```

The ~ (tilde) added to any suffix changes the file search into an SCCS file-name search, with the actual suffix named by the . (period) and all characters up to (but not including) the ~ (tilde). The **GFLAGS** macro passes flags to the SCCS to determine which SCCS file version to use.

The **make** program recognizes the following SCCS suffixes:

.C\~ C++ source
.c~ **c** source
.y~ **yacc** source grammar
.s~ Assembler source
.sh~ Shell
.h~ Header
.f~ FORTRAN
.l~ **lex** source

The **make** program has internal rules for changing the following SCCS files:

.C\~.a:

.C\~.c:

.C\~.o:

.c~:

.c~.a:

.c~.c:

.c~.o:

.f~:

.f~.a:

.f~.o:

.f~.f:

.h~.h:

.l~.o:

.s~.a:

.sh~:

.s~.o:

.y~.c:

.y~.o:

Description Files Stored in the Source Code Control System (SCCS)

If you specify a description file, or a file named **makefile** or **Makefile** is in the current directory, the **make** command does not look for a description file within SCCS. If a description file is not in the current directory and you enter the **make** command, the **make** program looks for an SCCS file named either **s.makefile** or **s.Makefile**. If either of these files are present, the **make** command uses a **get** command to direct SCCS to build the description file from that source file. When the SCCS generates the description file, the **make** command uses the file as a normal description file. When the **make** command finishes executing, it removes the created description file from the current directory.

Using the make Command with Non-Source Code Control System (SCCS) Files

Start the **make** program from the directory that contains the description file for the file to create. The variable name *desc-file* represents the name of that description file. Then, enter the command:

```
make -f desc-file
```

on the command line. If the name of the description file is `makefile` or `Makefile`, you do not have to use the **-f** flag. Enter macro definitions, flags, description file names, and target file names along with the **make** command on the command line as follows:

```
make [flags] [macro definitions] [targets]
```

The **make** program then examines the command-line entries to determine what to do. First, it looks at all macro definitions on the command line (entries that are enclosed in quotes and have equal signs in them) and assigns values to them. If the **make** program finds a definition for a macro on the command line different from the definition for that macro in the description file, it chooses the command-line definition for the macro.

Next, the **make** program looks at the flags. For more information, see the **make** command for a list of the flags that it recognizes.

The **make** program expects the remaining command-line entries to be the names of target files to be created. Any shell commands enclosed in back quotes that generate target names are performed by the **make** command. Then the **make** program creates the target files in left-to-right order. Without a target file name, the **make** program creates the first target file named in the description file that does not begin with a period. With more than one description file specified, the **make** command searches the first description file for the name of the target file.

How the make Command Uses the Environment Variables

Each time the **make** command runs, it reads the current environment variables and adds them to its defined macros. Using the **MAKEFLAGS** macro or the **MFLAGS** macro, the user can specify flags to be passed to the **make** command. If both are set, the **MAKEFLAGS** macro overrides the **MFLAGS** macro. The flags specified using these variables are passed to the **make** command along with any command-line options. In the case of recursive calls to the **make** command, using the **$(MAKE)** macro in the description file, the **make** command passes all flags with each invocation.

When the **make** command runs, it assigns macro definitions in the following order:

1. Reads the **MAKEFLAGS** environment variable.

   If the **MAKEFLAGS** environment variable is not present or null, the **make** command checks for a non-null value in the **MFLAGS** environment variable. If one of these variables has a value, the **make** command assumes that each letter in the value is an input flag. The **make** program uses these flags (except for the **-f**, **-p**, and **-d** flags, which cannot be set from the **MAKEFLAGS** or **MFLAGS** environment variable) to determine its operating conditions.
2. Reads and sets the input flags from the command line. The command line adds to the previous settings from the **MAKEFLAGS** or **MFLAGS** environment variable.
3. Reads macro definitions from the command line. The **make** command ignores any further assignments to these names.
4. Reads the internal macro definitions.
5. Reads the environment. The **make** program treats the environment variables as macro definitions and passes them to other shell programs.

Example of a Description File

The following example description file could maintain the **make** program. The source code for the **make** command is spread over a number of C language source files and a **yacc** grammar.

```
# Description file for the Make program
# Macro def: send to be printed
P = qprt
#  Macro def: source filenames used
    FILES = Makefile version.c defs main.c \
            doname.c misc.c files.c \
            dosy.c gram.y lex.c gcos.c
    # Macro def: object filenames used
    OBJECTS = version.o main.o doname.o \
              misc.o files.o dosys.o \
              gram.o
    # Macro def: lint program and flags
    LINT = lint -p
    # Macro def: C compiler flags
    CFLAGS = -O
    # make depends on the files specified
    # in the OBJECTS macro definition
    make:    $(OBJECTS)
    # Build make with the cc program
             cc $(CFLAGS) $(OBJECTS) -o make
    # Show the file sizes
             @size make
# The object files depend on a file
 # named defs
    $(OBJECTS):  defs
    # The file gram.o depends on lex.c
    # uses internal rules to build gram.o
    gram.o:  lex.c
    # Clean up the intermediate files
    clean:
             -rm *.o gram.c
             -du
    # Copy the newly created program
    # to /usr/bin and deletes the program
    # from the current directory
    install:
             @size make /usr/bin/make
             cp make /usr/bin/make ; rm make
    # Empty file "print" depends on the
    # files included in the macro FILES
    print:   $(FILES)
    # Print the recently changed files
             pr $? | $P
```

```
    # Change the date on the empty file,
    # print, to show the date of the last
    # printing
             touch print
# Check the date of the old
    # file against the date
    # of the newly created file
    test:
             make -dp | grep -v TIME >1zap
             /usr/bin/make -dp | grep -v TIME >2zap
             diff 1zap 2zap
             rm 1zap 2zap
    # The program, lint, depends on the
    # files that are listed
    lint:    dosys.c doname.c files.c main.c misc.c \
             version.c gram.c
    # Run lint on the files listed
    # LINT is an internal macro
             $(LINT) dosys. doname.c files.c main.c \
             misc.c version.c gram.c
             rm gram.c
    # Archive the files that build make
    arch:
             ar uv /sys/source/s2/make.a $(FILES)
```

The **make** program usually writes out each command before issuing it.

The following output results from entering the simple **make** command in a directory containing only the source and description file:

```
cc -O -c version.c
cc -O -c main.c
cc -O -c doname.c
cc -O -c misc.c
cc -O -c files.c
cc -O -c dosys.c
yacc  gram.y
mv y.tab.c gram.c
cc -O -c gram.c
cc version.o main.o doname.o misc.o files.o dosys.o
   gram.o -o make
make: 63620 + 13124 + 764 + 4951 = 82459
```

None of the source files or grammars are specified in the description file. However, the **make** command uses its suffix rules to find them and then issues the needed commands. The string of digits on the last line of the previous example results from the **size make** command. Because the @ (at sign) on the **size** command in the description file prevented writing of the command, only the sizes are written.

The output can be sent to a different printer or to a file by changing the definition of the **P** macro on the command line, as follows:

```
make print "P = print -sp"
```

OR

```
make print "P = cat >zap"
```

Chapter 15. m4 Macro Processor Overview

This chapter provides information about the **m4** macro processor, which is a front-end processor for any programming language being used in the operating system environment.

The **m4** macro processor is useful in many ways. At the beginning of a program, you can define a symbolic name or symbolic constant as a particular string of characters. You can then use the **m4** program to replace unquoted occurrences of the symbolic name with the corresponding string. Besides replacing one string of text with another, the **m4** macro processor provides the following features:

- Arithmetic capabilities
- File manipulation
- Conditional macro expansion
- String and substring functions

The **m4** macro processor processes strings of letters and digits called *tokens*. The **m4** program reads each alphanumeric token and determines if it is the name of a macro. The program then replaces the name of the macro with its defining text, and pushes the resulting string back onto the input to be rescanned. You can call macros with arguments, in which case the arguments are collected and substituted into the right places in the defining text before the defining text is rescanned.

The **m4** program provides built-in macros such as **define**. You can also create new macros. Built-in and user-defined macros work the same way.

Using the m4 Macro Processor

To use the **m4** macro processor, enter the following command:

```
m4 [file]
```

The **m4** program processes each argument in order. If there are no arguments or if an argument is **-** (dash), **m4** reads standard input as its input file. The **m4** program writes its results to standard output. Therefore, to redirect the output to a file for later use, use a command such as:

```
m4 [file] >outputfile
```

Creating a User-Defined Macro

define (*MacroName***,** *Replacement***)**	Defines new macro *MacroName* with a value of *Replacement*.

For example, if the following statement is in a program:

```
define(name, stuff)
```

The **m4** program defines the string name as `stuff`. When the string name occurs in a program file, the **m4** program replaces it with the string `stuff`. The string `name` must be ASCII alphanumeric and must begin with a letter or underscore. The string `stuff` is any text, but if the text contains parentheses the number of open, or left, parentheses must equal the number of closed, or right, parentheses. Use the **/** (slash) character to spread the text for `stuff` over multiple lines.

The open (left) parenthesis must immediately follow the word **define**. For example:

```
define(N, 100)
 . . .
if (i > N)
```

defines N to be 100 and uses the symbolic constant N in a later **if** statement.

Macro calls in a program have the following form:

```
name(arg1,arg2, . . . argn)
```

A macro name is recognized only if it is surrounded by nonalphanumerics. Using the following example:

```
define(N, 100)
 . . .
if (NNN > 100)
```

the variable NNN is not related to the defined macro N.

You can define macros in terms of other names. For example:

```
define(N, 100)
define(M, N)
```

defines both M and N to be 100. If you later change the definition of N and assign it a new value, M retains the value of 100, not N.

The **m4** macro processor expands macro names into their defining text as soon as possible. The string N is replaced by 100. Then the string M is also replaced by 100. The overall result is the same as using the following input in the first place:

```
define(M, 100)
```

The order of the definitions can be interchanged as follows:

```
define(M, N)
define(N, 100)
```

Now M is defined to be the string N, so when the value of M is requested later, the result is the value of N at that time (because the M is replaced by N, which is replaced by 100).

Using the Quote Characters

To delay the expansion of the arguments of **define**, enclose them in quote characters. If you do not change them, quote characters are ' ' (left and right single quotes). Any text surrounded by quote characters is not expanded immediately, but quote characters are removed. The value of a quoted string is the string with the quote characters removed. If the input is:

```
define(N, 100)
define(M, 'N')
```

the quote characters around the N are removed as the argument is being collected. The result of using quote characters is to define M as the string N, not 100. The general rule is that the **m4** program always strips off one level of quote characters whenever it evaluates something. This is true even outside of macros. To make the word define appear in the output, enter the word in quote characters, as follows:

```
'define' = 1;
```

Another example of using quote characters is redefining N. To redefine N, delay the evaluation by putting N in quote characters. For example:

```
define(N, 100)
. . .
define('N', 200)
```

To prevent problems from occurring, quote the first argument of a macro. For example, the following fragment does not redefine N:

```
define(N, 100)
. . .
define(N, 200)
```

The N in the second definition is replaced by 100. The result is the same as the following statement:

```
define(100, 200)
```

The **m4** program ignores this statement because it can only define names, not numbers.

Changing the Quote Characters

Quote characters are normally ' ' (left or right single quotes). If those characters are not convenient, change the quote characters with the following built-in macro:

changequote (*l***, *r*)** Changes the left and right quote characters to the characters represented by the *l* and *r* variables.

To restore the original quote characters, use **changequote** without arguments as follows:

```
changequote
```

Arguments

The simplest form of macro processing is replacing one string by another (fixed) string. However, macros can also have arguments, so that you can use the macro in different places with different results. To indicate where an argument is to be used within the replacement text for a macro (the second argument of its definition), use the symbol **$***n* to indicate the *n*th argument. When the macro is used, the **m4** macro processor replaces the symbol with the value of the indicated argument. For example, the symbol:

```
$2
```

refers to the second argument of a macro. Therefore, if you define a macro called bump as:

```
define(bump, $1 = $1 + 1)
```

the **m4** program generates code to increment the first argument by 1. The bump(x) statement is equivalent to x = x + 1.

A macro can have as many arguments as needed. However, you can access only nine arguments using the **$***n* symbol ($1 through $9). To access arguments past the ninth argument, use the **shift** macro.

shift (*ParameterList***)** Returns all but the first element of *ParameterList* to perform a destructive left shift of the list.

This macro drops the first argument and reassigns the remaining arguments to the **$***n* symbols (second argument to $1, third argument to $2. . . tenth argument to $9). Using the **shift** macro more than once allows access to all arguments used with the macro.

The **$0** macro returns the name of the macro. Arguments that are not supplied are replaced by null strings, so that you can define a macro that concatenates its arguments like this:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

Thus:

```
cat(x, y, z)
```

is the same as:

```
xyz
```

Arguments `$4` through `$9` in this example are null since corresponding arguments were not provided.

The **m4** program discards leading unquoted blanks, tabs, or new-line characters in arguments, but keeps all other white space. Thus:

```
define(a, b c)
```

defines `a` to be `b c`.

Arguments are separated by commas. Use parentheses to enclose arguments containing commas, so that the comma does not end the argument. For example:

```
define(a, (b,c))
```

has only two arguments. The first argument is `a`, and the second is `(b,c)`. To use a comma or single parenthesis, enclose it in quote characters.

Using a Built-In m4 Macro

The **m4** program provides a set of predefined macros. The subsequent sections explain many of the macros and their uses.

Removing a Macro Definition

undefine (′*MacroName*′**)** Removes the definition of a user-defined or built-in macro (′*MacroName*′)

For example:

```
undefine('N')
```

removes the definition of N. Once you remove a built-in macro with the **undefine** macro, as follows:

```
undefine('define')
```

then you cannot use its definition of the built-in macro again.

Single quotes are required in this case to prevent substitution.

Checking for a Defined Macro

ifdef (′*MacroName*′**,** *Argument1*, *Argument2***)**

If macro *MacroName* is defined and is not defined to zero, returns the value of *Argument1*. Otherwise, it returns *Argument2*.

The **ifdef** macro permits three arguments. If the first argument is defined, the value of ifdef is the second argument. If the first argument is not defined, the value of ifdef is the third argument. If there is no third argument, the value of ifdef is null.

Using Integer Arithmetic

The **m4** program provides the following built-in functions for doing arithmetic on integers only:

incr (*Number***)**	Returns the value of *Number* + 1.
decr (*Number* **)**	Returns the value of *Number* - 1.
eval	Evaluates an arithmetic expression.

Thus, to define a variable as one more than the *Number* value, use the following:

```
define(Number, 100)
define(Number1, 'incr(Number)')
```

This defines `Number1` as one more than the current value of `Number`.

The **eval** function can evaluate expressions containing the following operators (listed in decreasing order of precedence):

unary + and -

**** or ^ (exponentiation)**

*** / % (modulus)**

+ -

== != < <= > >=

!(not)

& or && (logical AND)

| or || (logical OR)

Use parentheses to group operations where needed. All operands of an expression must be numeric. The numeric value of a true relation (for example, 1 > 0) is 1, and false is 0. The precision of the **eval** function is 32 bits.

For example, define `M` to be `2==N+1` using the **eval** function as follows:

```
define(N, 3)
define(M, 'eval(2==N+1)')
```

Use quote characters around the text that defines a macro unless the text is very simple.

Manipulating Files

To merge a new file in the input, use the built-in **include** function.

include (*File***)**	Returns the contents of the file *File*.

For example:

```
include(FileName)
```

inserts the contents of `FileName` in place of the **include** command.

A fatal error occurs if the file named in the **include** macro cannot be accessed. To avoid a fatal error, use the alternate form **sinclude**.

sinclude (*File* **)** Returns the contents of the file *File*, but does not report an error if it cannot access *File*.

The **sinclude** (silent include) macro does not write a message, but continues if the file named cannot be accessed.

Redirecting Output

The output of the **m4** program can be redirected again to temporary files during processing, and the collected material can be output upon command. The **m4** program maintains nine possible temporary files, numbered 1 through 9. If you use the built-in **divert** macro.

divert (*Number***)** Changes output stream to the temporary file *Number*.

The **m4** program writes all output from the program after the **divert** function at the end of temporary file, *Number*. To return the output to the display screen, use either the **divert** or **divert(0)** function, which resumes the normal output process.

The **m4** program writes all redirected output to the temporary files in numerical order at the end of processing. The **m4** program discards the output if you redirect the output to a temporary file other than 0 through 9.

To bring back the data from all temporary files in numerical order, use the built-in **undivert** macro.

undivert (*Number1***,** *Number2*... **)** Appends the contents of the indicated temporary files to the current temporary file.

To bring back selected temporary files in a specified order, use the built-in **undivert** macro with arguments. When using the **undivert** macro, the **m4** program discards the temporary files that are recovered and does not search the recovered data for macros.

The value of the **undivert** macro is not the diverted text.

divnum Returns the value of the currently active temporary file.

If you do not change the output file with the **divert** macro, the **m4** program puts all output in a temporary file named 0.

Using System Programs in a Program

You can run any program in the operating system from a program by using the built-in **syscmd** macro. For example, the following statement runs the **date** program:

```
syscmd(date)
```

Using Unique File Names

Use the built-in **maketemp** macro to make a unique file name from a program.

maketemp (*String...nnnnn...String***)** Creates a unique file name by replacing the characters *nnnnn* in the argument string with the current process ID.

For example, for the statement:

```
maketemp(myfilennnnn)
```

the **m4** program returns a string that is `myfile` concatenated with the process ID. Use this string to name a temporary file.

Using Conditional Expressions

ifelse (*String1***, *String2*, *Argument1*, *Argument2*)**

If *String1* matches *String2*, returns the value of *Argument1*. Otherwise it returns *Argument2*.

The built-in **ifelse** macro performs conditional testing. In the simplest form:

```
ifelse(a, b, c, d)
```

compares the two strings `a` and `b`.

If `a` and `b` are identical, the built-in **ifelse** macro returns the string `c`. If they are not identical, it returns string `d`. For example, you can define a macro called `compare` to compare two strings and return `yes` if they are the same, or `no` if they are different, as follows:

```
define(compare, 'ifelse($1, $2, yes, no)')
```

The quote characters prevent the evaluation of the **ifelse** macro from occurring too early. If the fourth argument is missing, it is treated as empty.

The **ifelse** macro can have any number of arguments, and therefore, provides a limited form of multiple-path decision capability. For example:

```
ifelse(a, b, c, d, e, f, g)
```

This statement is logically the same as the following fragment:

```
if(a == b) x = c;
else if(d == e) x = f;
else x = g;
return(x);
```

If the final argument is omitted, the result is null, so:

```
ifelse(a, b, c)
```

is `c` if `a` matches `b`, and null otherwise.

Manipulating Strings

len Returns the byte length of the string that makes up its argument

Thus:

```
len(abcdef)
```

is 6, and:

```
len((a,b))
```

is 5.

dlen Returns the length of the displayable characters in a string

Characters made up from 2-byte codes are displayed as one character. Thus, if the string contains any 2-byte, international character-support characters, the results of **dlen** will differ from the results of **len**.

substr (*String***,** *Position***,** *Length***)** Returns a substring of *String* that begins at character number *Position* and is *Length* characters long.

Using input, **substr (***s***,** *i***,** *n***)** returns the substring of *s* that starts at the *i*th position (origin zero) and is *n* characters long. If *n* is omitted, the rest of the string is returned. For example, the function:

```
substr('now is the time',1)
```

returns the following string:

```
now is the time
```

index (*String1***,** *String2***)** Returns the character position in *String1* where *String2* starts **(**starting with character number 0), or -1 if *String1* does not contain *String2*.

As with the built-in **substr** macro, the origin for strings is 0.

translit (*String***,** *Set1***,** *Set2***)** Searches *String* for characters that are in *Set1*. If it finds any, changes (transliterates) those characters to corresponding characters in *Set2*.

It has the general form:

```
translit(s, f, t)
```

which modifies `s` by replacing any character found in `f` by the corresponding character of `t`. For example, the function:

```
translit(s, aeiou, 12345)
```

replaces the vowels by the corresponding digits. If `t` is shorter than `f`, characters that do not have an entry in `t` are deleted. If `t` is not present at all, characters from `f` are deleted from `s`. So:

```
translit(s, aeiou)
```

deletes vowels from string `s`.

dnl Deletes all characters that follow it, up to and including the new-line character.

Use this macro to get rid of empty lines. For example, the function:

```
define(N, 100)
define(M, 200)
define(L, 300)
```

results in a new-line at the end of each line that is not part of the definition. These new-line characters are passed to the output. To get rid of the new lines, add the built-in **dnl** macro to each of the lines.

```
define(N, 100) dnl
define(M, 200) dnl
define(L, 300) dnl
```

Printing

errprint (*String***)** Writes its argument (*String*) to the standard error file

For example:

```
errprint ('error')
```

dumpdef ('*MacroName*'... **)**	Dumps the current names and definitions of items named as arguments ('*MacroName*'...)

If you do not supply arguments, the **dumpdef** macro prints all current names and definitions. Remember to quote the names.

List of Additional m4 Macros

A list of additional **m4** macros, with a brief explanation of each, follows:

changecom (*l***,** *r* **)**	Changes the left and right comment characters to the characters represented by the *l* and *r* variables.
defn (*MacroName***)**	Returns the quoted definition of *MacroName*
en (*String***)**	Returns the number of characters in *String*.
eval (*Expression***)**	Evaluates *Expression* as a 32-bit arithmetic expression.
m4exit (*Code***)**	Exits **m4** with a return code of *Code*.
m4wrap (*MacroName***)**	Runs macro *MacroName* at the end of **m4**.
popdef (*MacroName***)**	Replaces the current definition of *MacroName* with the previous definition saved with the **pushdef** macro.
pushdef (*MacroName***,** *Replacement***)**	Saves the current definition of *MacroName* and then defines *MacroName* to be *Replacement*.
syscmd (*Command***)**	Executes the system command *Command* with no return value.
sysval	Gets the return code from the last use of the **syscmd** macro.
traceoff (*MacroList***)**	Turns off trace for any macro in *MacroList*. If *MacroList* is null, turns off all tracing.
traceon (*MacroName***)**	Turns on trace for macro *MacroName*. If *MacroName* is null, turns trace on for all macros.

Chapter 16. National Language Support

National Language Support (NLS) provides commands and library subroutines for a single worldwide system base.

- Code sets
- Character classifications
- Character comparison rules
- Character collation order
- Numeric and monetary formatting
- Date and time formatting
- Message-text language

NLS Capabilities

An application that runs in an international environment must not have built-in assumptions about:

- "Locale-Specific and Culture-Specific Conventions"
- "User Messages in Native Languages"
- "Code Set Support"
- "Input Method Support" on page 330

This information must be determined during application execution. NLS provides these capabilities and a base upon which new languages and code sets can be supported. As a result, programs can be ported across national language and locale boundaries. The POSIX.1 standard, the POSIX.2 standard, the ANSI/ISO C language standard, and the X/Open XPG specifications define standards for providing NLS support.

Locale-Specific and Culture-Specific Conventions

An internationalized program can process information correctly for different locations. (For example, the conventions for specifying date and time differ in the United States and England.) Similarly, the decimal point (radix character) and monetary symbols differ between the two countries. These types of language and cultural conventions for handling information are defined in a *locale*. For more information about locales, see "Locale Overview for Programming" on page 330.

User Messages in Native Languages

To facilitate translations of messages into various languages and make translated messages available to the program based on a user's locale, messages are kept separate from the programs by providing them in the form of message catalogs that a program can access at run time. To aid in this task, commands and subroutines are provided by the Message Facility. For more information, see "Message Facility Overview for Programming" on page 480.

Code Set Support

A character is any symbol used for the organization, control, or representation of data. A group of such symbols for describing a particular language make up a character set. A code set contains the encoding values for a character set. The encoding values in a code set provide the interface between the system and its input and output devices.

In the past, the effort was directed at encoding the English alphabet. A 7-bit encoding method was adequate for this purpose because the number of English characters is not large. The C language defined

the **char** data type to indicate a 7-bit character. A byte is an 8-bit quantity and is therefore used to represent a **char** data type value. The eighth bit was typically used for parity.

To support larger character sets, such as the Asian languages (for example, Chinese, Japanese, and Korean), additional code sets were developed that contained multibyte encodings. Because of multibyte encodings, the old concept of the **char** data type is no longer sufficient to represent a character. The C standard continues to refer to the **char** data type to mean a 7-bit character. However, the **char** data type really means a byte, either signed or unsigned.

An internationalized program must accurately read data generated in different code set environments and process the information accurately. You can use **nl_langinfo(CODESET)** to obtain the current code set in a process. The return value is a **char** pointer that is the name of the code set in the system. Because code set names are not standard, programs should not depend on any specific value for this string. Knowing the current code set can aid in code-set conversion. NLS supplies converters that translate character encoding values found in different code sets. For more information, see “Converters Overview for Programming” on page 410.

Input Method Support

The input of characters becomes complicated for languages having large character sets. For example, in Chinese, Korean, and Japanese, where the number of characters is large, it is not possible to provide one-to-one key mapping for a keystroke to a character. However, a special input method enables the user to enter phonetic or stroke characters and have them converted into native-language characters. A keyboard map associated with each keyboard matches sequences of one or more keystrokes with the appropriate character encoding. For more information, see the “Input Method Overview” on page 452.

Overview of Chapter Contents

This NLS chapter contains the following information:

Locale Overview for Programming

National Language Support (NLS) provides commands and library subroutines for a single worldwide system base. An internationalized system has no built-in assumptions or dependencies on language-specific or cultural-specific conventions. All locale information is obtained at program run time.

The following concepts are needed to understand the internationalization of programs:

Working with Code Sets

ASCII is a code set containing 128 code points (0x00 through 0x7F). The ASCII character set contains control characters, punctuation marks, digits, and the uppercase and lowercase English alphabet. Several 8-bit code sets incorporate ASCII as a proper subset. However, throughout this document, ASCII refers to 7-bit-only code sets. To emphasize this, it is referred to as 7-bit ASCII. The 7-bit ASCII code set is a proper subset of all supported code sets and is referred to as the *portable character set*. For more information, see “Code Set Overview” on page 379 .

Single-Byte and Multibyte Code Sets

A single-byte encoding method is sufficient for representing the English character set because the number of characters is not large. To support larger alphabets, such as Japanese and Chinese, additional code sets containing multibyte encodings are necessary. All supported single-byte and multibyte code sets contain the single-byte ASCII character set. Therefore, programs that handle multibyte code sets must handle character encodings of one or more bytes.

Examples of single-byte code sets are the ISO 8859 family of code sets and the IBM-850 code set. Examples of multibyte character sets are the IBM-eucJP and the IBM-943 code sets. The single-byte code sets have at most 256 characters and the multibyte code sets have more than 256 (without any theoretical limit).

The Unique Code-Point Range

None of the supported code sets have bytes 0x00 through 0x3F in any byte of a multibyte character. This group of code points is called the *unique code-point range*. Furthermore, these code points always refer to the same characters as specified for 7-bit ASCII. This is a special property governing all supported code sets. ASCII Characters in the Unique Code-Point Range (“ASCII Characters” on page 380) lists the characters in the unique code-point range.

For more information about code sets, see the “Code Set Overview” on page 379.

Data Representation

Because the encoding for some characters requires more than one byte, a single character may be represented by one or several bytes when data is created in files or transferred between a computer and its I/O devices. This external representation of data is referred to as the *file code* or *multibyte character code* representation of a character.

For processing strings of such characters, it is more efficient to convert file codes into a uniform representation. This converted form is intended for internal processing of characters. This internal representation of data is referred to as the *process code* or *wide character code* representation of the character. An understanding of multibyte character and wide character codes is essential to the overall internationalization strategy.

Multibyte Character Code Data Representation

A multibyte character code is an external representation of data, regardless of whether it is character input from a keyboard or a file on a disk. Within the same code set, the number of bytes that represent the multibyte code of a character can vary. You must use NLS functions for character processing to ensure code set independence.

For example, a code set may specify the following character encodings:

```
C  = 0x43
*  = 0x81 0x43
*C = 0x81 0x43& 0x43
```

A program searching for `C`, not accounting for multibyte characters, finds the second byte of the `*C` string and assumes it found `C` when in fact it found the second byte of the `*` (asterisk) character.

Wide Character Code Data Representation

The wide character code was developed so that multibyte characters could be processed more efficiently internally in the system. A multibyte character representation is converted into a uniform internal representation (wide character code) so that internally all characters have the same length. Using this internal form, character processing can be done in a code set-independent fashion. The wide character code refers to this internal representation of characters.

The **wchar_t** data type is used to represent the wide character code of a character. The size of the **wchar_t** data type is implementation-specific. It is a **typedef** definition and can be found in the **ctype.h**, **stddef.h**, and **stdlib.h** files. No program should assume a particular size for the **wchar_t** data type, enabling programs to run under implementations that use different sizes for the **wchar_t** data type.

On AIX 4.3, the **wchar_t** datatype is implemented as an unsigned short value (16 bits). The locale methods in AIX have been standardized such that in most locales, the value stored in the **wchar_t** for a particular character will always be its Unicode data value. For applications which are intended to run only on AIX, this allows certain applications handle the **wchar_t** datatype in a consistent fashion, even if the underlying codeset is unknown. All locales on AIX 4.3 will use Unicode for their wide character code values (process code), except the following:

1. Locales based on the IBM-850 codeset are provided strictly for compatibility with previous releases of AIX. These locales have not been modified from previous releases and will be removed in a future release of AIX. It is strongly suggested that users use the industry standard ISO8859-1 codeset instead of IBM-850. For IBM-850 locales, the **wchar_t** data value will be the same value as the IBM-850 codepoint value.
2. The IBM-eucTW codeset (LANG =**zh_TW**) contains many characters that are not contained in the Unicode standard. Because of this, it is impossible to represent these characters with a Unicode wide character value. Applications that need to have Unicode based **wchar_t** data for Traditional Chinese should use the **Zh_TW** locale (big5 codeset) instead.

Character Properties

Every character has several language-dependent attributes or properties. These properties are called *class properties*. For example, the lowercase letter `a` in U.S. English has the following properties:

- alphabetic
- hexadecimal digit
- printable
- lowercase
- graphic

Character class properties are specified by the **LC_CTYPE** category.

Collation-Order Properties

Character ordering or *collation* refers to the culture-specific ordering of characters. This ordering differs from that based on the ordinal value of a character in a code set. Collation-based ordering is dependent on the language. Character collation is specified by the **LC_COLLATE** category. The term *collating element* refers to one or more characters that have a collation value in a specific locale. The Spanish ll character is an example of a multicharacter collating element.

To sort the characters in any given language in the proper order, a Weight is assigned to each character so they sort as expected. However, a character's sort value and code-point value are not necessarily related.

One set of weights is not sufficient to sort strings for all languages. For example, in the case of the German words b<*a-umlaut*>ch and bane, if there is only one set of weights, and the weight of the letter a is less than that of <*a-umlaut*>, then bane sorts before b<*a-umlaut*>ch. However, the opposite result is correct. To satisfy the requirement of this example, two sets of weights, the Primary and Secondary Weights, are given to each character in the language. In the case of the characters a and <*a-umlaut*>, they have the same Primary Weights, but differ in their Secondary Weights. In the German locale, the Secondary Weight of a is less than that of <*a-umlaut*>.

The sorting algorithm first compares the two strings based on the Primary Weights of each character. If the Primary Weight values are the same, the two strings are compared again based on their Secondary Weights. In this example, the Primary Weights of the first two characters ba and b<*a-umlaut*> are the same, but the Primary Weights of the characters that follow (c and n, respectively) differ. As a result of this comparison, b<*a-umlaut*>ch is sorted before bane.

Here, the Secondary Weights are not used to collate the strings. However, as in the case of the strings bach and b<*a-umlaut*>ch, Secondary Weights must be used to get the proper order. When compared using Primary Weight values, these two strings are found to be equivalent. To break the tie, the Secondary Weights of a and <*a-umlaut*> are used. Because the Secondary Weight of a is less than that of <*a-umlaut*>, the string bach sorts before b<*a-umlaut*>ch.

Characters having the same Primary Weights belong to the same *equivalence class*. In this example, the characters a and <*a-umlaut*> are said to be members of the same equivalence class.

In string collation, each pair of strings is first compared based on Primary Weight. If the two strings are equal, they are compared again based on their Secondary Weights. If still equal, they are compared again based on Tertiary Weights up to the limit set by the **COLL_WEIGHTS_MAX** collating weight limit specified in the **sys/limits.h** file.

Code-Set Width

Code-set width refers to the maximum number of bytes required to represent a character as a file code. This information is specified by the **LC_CTYPE** category.

Code-Set Display Width

Code-set display width refers to the maximum number of columns required to display a character on a terminal. This information is specified by the **LC_CTYPE** category.

Localization

An internationalized program must process information correctly for different locations. For example, in the United States, the date format 9/6/1990 is interpreted to mean the sixth day of the ninth month of the year

1990. The United Kingdom interprets the same date format to mean the ninth day of the sixth month of the year 1990. The formatting of numerical and monetary data is also country specific, as in the case of the U.S. dollar and the U.K. pound.

A *locale* is defined by language-specific and cultural-specific conventions for processing information. All such information should be accessible to a program at run time so that the same program can display or process data differently for different countries. The process of providing a language interface to obtain and process this information into a database containing the locale-specific data is known as *localization*.

The **setlocale** subroutine establishes locale information. This subroutine uses the values of certain environment variables to initialize locale information contained in locale definition files. To deal with locale data in a logical manner, locale definition source files are divided into six categories defining specific aspects of the locale data.

Locale Categories

A *category* is a group of language-specific and culture-specific data. For instance, data referring to date and time formatting, the names of the days of the week, and names of the months is grouped into the **LC_TIME** category. Each category uses a set of keywords that describe a particular aspect of a locale. The following standard categories can be defined in a locale definition source file:

LC_COLLATE	Defines string-collation order information.
LC_CTYPE	Defines character classification, case conversion, and other character attributes.
LC_MESSAGES	Defines the format for affirmative and negative responses.
LC_MONETARY	Defines rules and symbols for formatting monetary numeric information.
LC_NUMERIC	Defines rules and symbols for formatting nonmonetary numeric information.
LC_TIME	Lists rules and symbols for formatting time and date information.

Understanding Locale

Locale information consists of data from these six categories. Each locale is described by a locale definition file. These files are named by the language, territory and code set information they describe. The format for naming a locale definition file is:

language[_*territory*][.*codeset*][@*modifier*]

For example, the locale for the Danish language spoken in Denmark using the ISO8859-1 code set is da_DK.ISO8859-1. The da stands for the Danish language and the DK stands for Denmark. The short form of da_DK is sufficient to indicate this locale. The same language and territory using the IBM-850 code set is indicated by either Da_DK.IBM-850 or Da_DK for short.

The C or POSIX Locale

This locale refers to the ANSI C or POSIX-defined standard for the locale inherited by all processes at startup time. The C or POSIX locale assumes the 7-bit ASCII character set and defines information for the six previous categories.

The Installation Default Locale

The installation default locale refers to the locale selected at system installation time as the systemwide locale. For example, a French user in Canada may define the default locale to be fr_CA.ISO8859-1 (fr for French, CA for Canada, and ISO8859-1 for the code set). Every process uses this locale unless the NLS environment variables are changed.

NLS Environment Variables

For localization, NLS uses the following environment variables:

- **LANG**
- **LC_ALL**
- **LC_COLLATE**
- **LC_CTYPE**
- **LC_MESSAGES**
- **LC_MONETARY**
- **LC_NUMERIC**
- **LC_TIME**
- **LOCPATH**
- **NLSPATH**

The **LC_COLLATE**, **LC_CTYPE**, **LC_MONETARY**, **LC_NUMERIC**, **LC_TIME**, and **LC_MESSAGES** environment variables determine the current values for their respective categories.

The **LC_ALL** and **LANG** environment variables also determine the current locale.

The **NLSPATH** environment variable specifies a colon-separated list of directory names where the message catalog files are located. This environment variable is used by the Message Facility component of the NLS subsystem.

The **LOCPATH** environment variable specifies the directories where localization information such as locale database files, input method files, and iconv converters are located. This variable specifies a colon-separated list of directory names. The list is used for setting up the locale for a particular process.

Note: All **setuid** and **setgid** programs will ignore the **LOCPATH** environment variable.

The environment variables that affect locale information can be grouped into three priority classes:

Priority Class	Environment Variable
high	**LC_ALL**
medium	**LC_COLLATE**, **LC_CTYPE**, **LC_MESSAGES**, **LC_MONETARY**, **LC_NUMERIC**
low	**LANG**

When a locale is requested by the **setlocale** subroutine for a particular category or for all categories, the environment variable settings are queried by their priority level in the following manner:

- If the **LC_ALL** environment variable is set, all six categories use the locale it specified. For example, if the **LC_ALL** environment variable is equal to `en_US` and the **LANG** environment variable is equal to `fr_FR`, a call to the **setlocale** subroutine sets each of the six categories to the `en_US` locale.
- If the **LC_ALL** environment variable is not set, each individual category uses the locale specified by its corresponding environment variable. For example, if the **LC_ALL** environment variable is not set, the **LC_COLLATE** environment variable is set to `de_DE`, and the **LC_TIME** environment variable is set to `fr_CA`, then a call to the **setlocale** subroutine sets the **LC_COLLATE** category to `de_DE` and the **LC_TIME** category to `fr_CA`. Neither environment variable has precedence over the other in this situation.
- If the **LC_ALL** environment variable is not set, and a value for a particular **LC_*** environment variable is not set, the value of the **LANG** environment variable determines the setting for that specific category. For example, if the **LC_ALL** environment variable is not set, the **LC_CTYPE** environment variable is set to `en_US`, the **LC_NUMERIC** environment variable is not set, and the **LANG** environment variable is set to `is_IS`, then a call to the **setlocale** subroutine sets the **LC_CTYPE** category to `en_US` and the **LC_NUMERIC** category to `is_IS`. The **LANG** environment variable specifies the locale for only those categories not previously determined by an **LC_*** environment variable.

- If the **LC_ALL** environment variable is not set, a value for a particular **LC_*** environment variable is not set, and the value of the **LANG** environment variable is not set, the locale for that specific category defaults to the C locale. For example, if the **LC_ALL** environment variable is not set, the **LC_MONETARY** environment variable is set to sv_SE, the **LC_TIME** environment variable is not set, and the **LANG** environment variable is not set, then a call to the **setlocale** subroutine sets the **LC_MONETARY** category to sv_SE and the **LC_TIME** category to C.

Environment Variables Precedence Example

The following table shows the current setting of the environment variables and the effect of calling **setlocale**(**LC_ALL,**"") . After the **setlocale** subroutine is called, the string sorting and character properties are done as in the German language, the monetary formatting is done as in the US conventions, the numeric, time formatting is done in Danish conventions, the date and time data formatting is done in the Danish conventions, and the user messages are displayed in the Danish language. The last column indicates the locale setting after **setlocale**(**LC_ALL,**"") is called.

Environment Variable and Category Names	Value of Environment Variables	Value of Category After Call To setlocale(LC_ALL,"")
LC_COLLATE	de_DE	de_DE
LC_CTYPE	de_DE	de_DE
LC_MONETARY	en_US	en_US
LC_NUMERIC	(unset)	da_DK
LC_TIME	(unset)	da_DK
LC_MESSAGES	(unset)	da_DK
LC_ALL	(unset)	(not applicable)
LANG	da_DK	(not applicable)

Multibyte Subroutines

Multibyte subroutines process characters in file-code form. The names of these subroutines usually start with the prefix **mb**. However, some multibyte subroutines do not have this prefix. For example, the **strcoll** and **strxfrm** subroutines process characters in their multibyte form but do not have the **mb** prefix. The following standard C subroutines operate on bytes and can be used in handling multibyte data: **strcmp**, **strcpy**, **strncmp**, **strncpy**, **strcat**, and **strncat**. The standard C search subroutines **strchr**, **strrchr**, **strpbrk**, **strcspn**, **strrchr**, **strspn**, **strstr**, and **strtok** can be used in the following cases:

- Searching or scanning for characters in single-byte code sets
- Searching or scanning for unique code-point range characters in multibyte strings.

For more information about multibyte character subroutines, see “National Language Support Subroutines Overview” on page 339.

Wide Character Subroutines

Wide character subroutines process characters in process-code form. Wide character subroutines usually start with a **wc** prefix. However, there are exceptions to this rule. For example, the wide character classification functions use an **isw** prefix. To determine if a subroutine is a wide character subroutine, check if the subroutine prototype defines characters as **wchar_t** data type or **wchar_t** data pointer, or else check whether the subroutine returns a **wchar_t** data type. There are some exceptions to this rule. For example, the wide character classification subroutines accept **wint_t** data type values.

For more information about wide character subroutines, see “National Language Support Subroutines Overview” on page 339.

Bidirectionality and Character Shaping

An internationalized program may need to handle bidirectionality of text and character shaping.

Bidirectionality (BIDI) occurs when texts of different direction orientation appear together. For example, English text is read from left to right. Hebrew text is read from right to left. If both English and Hebrew texts appear on the same line, the text is bidirectional.

Character shaping occurs when the shape of a character is dependent on its position in a line of text. In some languages, such as Arabic, characters have different shapes depending on their position in a string and on the surrounding characters.

For more information about bidirectionality and character shaping, see “Layout (Bidirectional Text and Character Shaping) Overview” on page 373, “Character Shaping” on page 376, and “Introducing Layout Library Subroutines” on page 377.

Code Set Independence

The system needs certain information about code sets to communicate with the external environment. This information is hidden by the code set-independent library subroutines (NLS library). These subroutines pass information to the code set-dependent functions. Because NLS subroutines handle the necessary code set information, you do not need explicit knowledge of any code set when you write programs that process characters. This programming technique is called *code set independence*.

Determining Maximum Number of Bytes in Code Sets

You can use the **MB_CUR_MAX** macro to determine the maximum number of bytes in a multibyte character for the code set in the current locale. The value of this macro is dependent on the current setting of the **LC_CTYPE** category. Because the locale can differ between processes, running the **MB_CUR_MAX** macro in different processes or at different times may produce different results. The **MB_CUR_MAX** macro is defined in the **stdlib.h** header file.

You can use the **MB_LEN_MAX** macro to determine the maximum number of bytes in any code set that is supported by the system. This macro is defined in the **limits.h** header file.

Determining Character and String Display Widths

The **_max_disp_width** macro is operating-system-specific, and its use should be avoided in portable applications. If portability is not important, you can use the **_max_disp_width** macro to determine the maximum number of display columns required by a single character in the code set in the current locale . The value of this macro is dependent on the current setting of the **LC_CTYPE** category. If the value of this is 1 (one), all characters in the current code set require only one display column width on output.

When both **MB_CUR_MAX** and **_max_disp_width** are set to 1 (one), you can use the **strlen** subroutine to determine the display column width needed for a string. When **MB_CUR_MAX** is greater than one, use the **wcswidth** subroutine to find the display column width of the string.

The **wcswidth** and **wcwidth** wide character display width subroutines do not have corresponding multibyte functions. The **wcswidth** subroutine does not indicate how many characters can be displayed in the space available on a display. The **wcwidth** subroutine is useful for this purpose. This subroutine must be called repeatedly on a wide character string to find out how many characters can be displayed in the available positions on the display.

Exceptions to Code Set Knowledge: Unique Code-Point Range

There is one exception to the statement: ″No knowledge of the underlying code set can be assumed in a program.″ This exception arises due to the way the supported code sets are organized.

When a multibyte character string is searched for any character within the unique code-point range (for example, the . (period) character), it is not necessary to convert the string to process code form. It is sufficient to just look for that character (.) by examining each byte. This exception enables the kernel and utilities to search for the special characters . and / while parsing file names. If a program searches for any of the characters in the unique code-point range, the standard string functions that operate on bytes (such as **strchr**), should be used. “ASCII Characters” on page 380 lists the characters in the unique code-point range.

> **Note:** This exception is not a property that is applicable to all code sets. It should not be construed that this exception will remain valid in future releases.

File Name Matching

POSIX.2 defines the **fnmatch** subroutine to be used for file name matching. An application can use the **fnmatch** subroutine to read a directory and apply a pattern against each entry. For example, the **find** utility can use the **fnmatch** subroutine. The **pax** utility can use the **fnmatch** subroutine to process its pattern operands. Applications that need to match strings in a similar fashion can use the **fnmatch** subroutine.

Radix Character Handling

Note that the radix character, as obtained by **nl_langinfo(**RADIXCHAR**),** is a pointer to a string. It is possible that a locale may specify this as a multibyte character or as a string of characters. However, in AIX, a simplifying assumption is made that the RADIXCHAR is a single-byte character.

Programming Model

The programming model presented here highlights changes you need to make when an existing program is internationalized or when a new program is developed:

- Provide complete internationalization. Do not assume that characters have any specific properties. Determine the properties dynamically by using the appropriate interfaces. Do not assume properties of code sets, except for the ASCII characters with code points in the unique code-point range.
- Make programs code set-independent. Programs should not assume single-byte, double-byte, or multibyte encoding of any sort. Data can be processed in either process-code or file-code form by using the appropriate subroutines.
- Provide interaction with the kernel in file-code form only. The kernel does not handle process codes.
- The NLS subroutine library can handle processing based on file-code as well as processing based on process-code.

  > **Note:** Several subroutines based on process-code do not have corresponding subroutines based on file-code. Due to this asymmetry, it may be necessary to convert strings to process-code form and invoke the appropriate process-code subroutines.
- Some libraries may not provide processing in process-code form. An application needing these libraries must use file-codes when invoking functions from them.
- Programs can process characters either in process-code form or file-code form. It is possible to write code set-independent programs using both methods.

National Language Support Subroutines Overview

When internationalizing programs using National Language Support (NLS), it is important that there be some guidelines for providing this support. The intent of this section is to guide programmers in developing portable internationalized programs. An understanding of the concepts explained in “Locale Overview for Programming” on page 330 is a prerequisite to this section.

Introducing Locale Subroutines

Programs that perform locale-dependent processing, including user messages, must call the **setlocale** subroutine at the beginning of the program. This call should be the first executable statement in the **main** program. Programs that do not call the **setlocale** subroutine in this way inherit the C or POSIX locale. Such programs perform as in the C locale regardless of the setting of the **LC_*** and **LANG** environment variables.

Other subroutines are provided to determine the current settings for locale data formatting. For more information about these subroutines, see “Locale Subroutines” on page 340.

Introducing Time Formatting Subroutines

Programs that need to format or time into wide character code strings can use the **wcsftime** subroutine. Programs that need to convert multibyte strings into an internal time format can use the **strptime** subroutine. For more information about these subroutines, see “Time Formatting Subroutines” on page 345.

Introducing Monetary Formatting Subroutines

Programs that need to specify or access monetary quantities can call the **strfmon** subroutine. For more information about this subroutine, see “Monetary Formatting Subroutines” on page 346.

Introducing Multibyte and Wide Character Subroutines

The external representation of data is referred to as the *file code* representation of a character. When file code data is created in files or transferred between a computer and its I/O devices, a single character may be represented by one or several bytes. For processing strings of such characters, it is more efficient to convert these codes into a uniform-length representation. This converted form is intended for internal processing of characters. The internal representation of data is referred to as the *process code* or *wide character code* representation of the character.

NLS internationalization of programs is a blend of multibyte and wide character subroutines. A *multibyte* subroutine uses multibyte character sets. A *wide character* subroutine uses wide character sets. Multibyte subroutines have an **mb** prefix. Wide character subroutines have a **wc** prefix. The corresponding string-handling subroutines are indicated by the **mbs** and **wcs** prefixes, respectively. Deciding when to use multibyte or wide character subroutines can be made only after careful analysis.

If a program primarily uses multibyte subroutines, it may be necessary to convert the multibyte character codes to wide character codes to use certain wide character subroutines. If a program uses wide character subroutines, data may need to be converted to multibyte form for invoking subroutines. Both methods have drawbacks, depending on the program and the availability of standard subroutines to perform the required processing. For instance, there is no corresponding standard multibyte subroutine for the wide character display-column-width subroutine.

If a program can process its characters in multibyte code, this method should be used instead of converting the characters to wide character code.

For more information about the subroutines provided for converting between multibyte code and wide character code form, see “Multibyte Code and Wide Character Code Conversion Subroutines” on page 348.

wchar.h Header File

The **wchar.h** header file declares information necessary for programming with multibyte and wide character subroutines. The **wchar.h** header file declares the **wchar_t**, **wctype_t**, and **wint_t** data types, as well as several functions for testing wide characters. Because the number of characters implemented as wide characters exceeds that of basic characters, it is not possible to classify all wide characters into the existing classes used for basic characters. Therefore, it is necessary to provide a way of defining additional classes specific to some locale. The action of these subroutines is affected by the current locale.

The **wchar.h** header file also declares subroutines for manipulating wide character strings (that is, **wchar_t** data type arrays). Array length is always determined in terms of the number of **wchar_t** elements in an array. A null wide character code ends an array. A pointer to a **wchar_t** or void array always points to the initial element of the array.

> **Note:** If the number of **wchar_t** elements in an array exceeds the defined array length, unpredictable results can occur.

Introducing Internationalized Regular Expression Subroutines

Programs that contain internationalized regular expressions can use the **regcomp**, **regexec**, **regerror**, **regfree**, and **fnmatch** subroutines. For more information about these subroutines, see “Internationalized Regular Expression Subroutines” on page 370.

Locale Subroutines

The locale of a process determines the way character collation, character classification, date and time formatting, numeric punctuation, monetary punctuation, and message output are handled. The following section describes how to set and access information about the current locale in a program using National Language Support (NLS).

Setting the Locale

Every internationalized program must set the current locale using the **setlocale** subroutine. This subroutine allows a process to change or query the current locale by accessing locale databases.

When a process is started, its current locale is set to the C or POSIX locale. A program that depends on locale data not defined in the C or POSIX locale must invoke the **setlocale** subroutine in the following manner before using any of the locale-specific information:

```
setlocale(LC_ALL, "");
```

Accessing Locale Information

The following subroutines provide access to information defined in the current locale as determined by the most recent call to the **setlocale** subroutine:

localeconv	Provides access to locale information defined in the **LC_MONETARY** and **LC_NUMERIC** categories of the current locale. The **localeconv** subroutine retrieves information about these categories, places the information in a structure of type **lconv** as defined in the **locale.h** file, and returns a pointer to this structure.
nl_langinfo	Returns a pointer to a null-terminated string containing information defined in the **LC_CTYPE**, **LC_MESSAGES**, **LC_MONETARY**, **LC_NUMERIC**, and **LC_TIME** categories of the current locale.

rpmatch Tests for positive and negative responses. These are specified in the **LC_MESSAGES** category of the current locale. Responses can be regular expressions as well as simple strings. The **rpmatch** subroutine is not an industry-standard subroutine. Portable applications should not assume that this subroutine is available.

The **localeconv** and **nl_langinfo** subroutines do not provide access to all **LC_*** categories.

The current locale setting for a category can be obtained by: **setlocale**(*Category*, (char*)0). The return value is a string specifying the current locale for *Category*. The following example determines the current locale setting for the **LC_CTYPE** category:

```
char *ctype_locale; ctype_locale = setlocale(LC_CTYPE, (char*)0);
```

Examples

1. The following example uses the **setlocale** subroutine to change the locale from the default C locale to the locale specified by the environment variables, consistent with the hierarchy of the locale environment variables:

```
#include <locale.h>
main()
{

    char *p;

    p = setlocale(LC_ALL, "");

   /*
    **  The program will have the locale as set by the
    **  LC_* and LANG variables.
    */
 }
```

2. The following example uses the **setlocale** subroutine to obtain the current locale setting for the **LC_COLLATE** category:

```
#include <stdio.h>
#include <locale.h>

main()
{
    char *p;

    /*  set the current locale to what is specified  */
    p = setlocale(LC_ALL, "");
    /*  The current locale settings for all the
    **  categories is pointed to by p
    */

    /*
    **  Find the current setting for the
    **  LC_COLLATE category
    */
    p = setlocale(LC_COLLATE, NULL);
    /*
    **  p points to a string containing the current locale
    **  setting for the LC_COLLATE category.
    */

 }
```

3. The following example uses the **setlocale** subroutine to obtain the current locale setting and saves it for later use. This action allows the program to temporarily change the locale to a new locale. After processing is complete, the locale can be returned to its original state.

```
#include <stdio.h>
#include <locale.h>
#include <string.h>

#define NEW_LOCALE "MY_LOCALE"

main()
{
    char *p, *save_locale;

    p = setlocale(LC_ALL, "");
    /*
    **  Initiate locale. p points to the current locale
    **  setting for all the categories
    */

    save_locale = (char *)malloc(strlen(p) +1);
    strcpy(save_locale, p);
        /*  Save the current locale setting */
    p = setlocale(LC_ALL, NEW_LOCALE);
        /* Change to new locale */

    /*
    **  Do processing  ...
    */

     /*  Change back to old locale  */
    p = setlocale(LC_ALL, save_locale);  /* Restore old locale */

     free(save_locale);    /* Free the memory */
}
```

4. The following example uses the **setlocale** subroutine to set the **LC_MESSAGES** category to the locale determined by the environment variables. All other categories remain set to the C locale.

```
#include <locale.h>

main()
{
    char *p;

    /*
    **  The program starts in the C locale for all categories.
    */

    p = setlocale(LC_MESSAGES, "");

     /*
    **  At this time the LC_COLLATE, LC_CTYPE, LC_NUMERIC,
    **  LC_MONETARY, LC_TIME will be in the C locale.
    **  LC_MESSAGES will be set to the current locale setting
    **  as determined by the environment variables.
    */
}
```

5. The following example uses the **localeconv** subroutine to obtain the decimal-point setting for the current locale:

```
#include <locale.h>

main()
{
        struct lconv *ptr;
        char *decimal;

        (void)setlocale(LC_ALL, "");
        ptr = localeconv();
        /*
        ** Access the data obtained. For example,
```

```
        ** obtain the current decimal point setting.
        */
        decimal = ptr->decimal_point;
}
```

6. The following example uses the **nl_langinfo** subroutine to obtain the date and time format for the current locale:

```
#include <langinfo.h>
#include <locale.h>
main()
{
    char *ptr;
    (void)setlocale(LC_ALL, "");
    ptr = nl_langinfo(D_T_FMT);
}
```

7. The following example uses the **nl_langinfo** subroutine to obtain the radix character for the current locale:

```
#include <langinfo.h>
#include <locale.h>

main()
{
    char *ptr;
    (void)setlocale(LC_ALL, "");  /* Set the program's locale  */
    ptr = nl_langinfo(RADIXCHAR); /* Obtain the radix character*/
}
```

8. The following example uses the **nl_langinfo** subroutine to obtain the setting of the currency symbol for the current locale:

```
#include <langinfo.h>
#include <locale.h>

main()
{
    char *ptr;
    (void)setlocale(LC_ALL, "");  /* Set the program's locale  */
    ptr = nl_langinfo(CRNCYSTR);  /* Obtain the currency string*/
    /* The currency string will be "-$" in the U. S. locale. */
}
```

9. The following example uses the **rpmatch** subroutine to obtain the setting of affirmative and negative response strings for the current locale:

   The affirmative and negative responses as specified in the locale database are no longer simple strings; they can be regular expressions. For example, the yesexpr can be the following regular expression, which will accept an upper or lower case letter `y`, followed by zero or more alphabetic characters; or the character `0` followed by `K`. Thus, yesexpr may be the following regular expression:

   ```
   ([yY][:alpha:]*|OK)
   ```

   The standards do not contain a subroutine to retrieve and compare this information. You can use the AIX-specific **rpmatch(const char *response)** subroutine.

```
#include <stdio.h>
#include <langinfo.h>
#include <locale.h>
#include <regex.h>

int rpmatch(const char *);
    /*
    **  Returns 1 if yes response, 0 if no  response,
    ** -1 otherwise
    */

main()
{
    int  ret;
```

```
    char *resp;

(void)setlocale(LC_ALL, "");

    do {
        /*
        ** Obtain the response to the query for yes/no strings.
        ** The string pointer resp points to this response.
        ** Check if the string is yes.
        */
    ret = rpmatch(resp);

    if(ret == 1){
        /* Response was yes. */
        /* Process accordingly. */
        }else if(ret == 0){
        /* Response was negative. */
        /* Process negative response. */
        }else if(ret<0){
        /* No match with yes/no occurred. */
        continue;
        }
    }while(ret <0);
}
```

10. The following example provides a method of implementing the **rpmatch** subroutine. Note that most applications should use the **rpmatch** subroutine in **libc**. The following implementation of **rpmatch** is just for illustration purposes.

    Note that **nl_langinfo(YESEXPR)** and **nl_langinfo(NOEXPR)** are used to obtain the regular expressions for the affirmative and negative responses respectively.

```
#include <langinfo.h>
#include <regex.h>
/*
** rpmatch() performs comparison of a string to a regular expression
** using the POSIX.2 defined regular expression compile and match
** functions. The first argument is the response from the user and the
** second string is the current locale setting of the regular expression.
*/
int rpmatch( const char *string)


{
     int status;
     int retval;
     regex_t re;
     char *pattern;

     pattern = nl_langinfo(YESEXPR);
     /* Compile the regular expression pointed to by pattern. */
     if( ( status = regcomp( &re, pattern, REG_EXTENDED | REG_NOSUB )) != 0 ){
          retval = -2; /*-2 indicates yes expr compile error */
          return(retval);
     }
     /* Match the string with the compiled regular expression. */
     status = regexec( &re, string, (size_t)0, (regmatch_t *)NULL, 0);
     if(status == 0){
          retval = 1;   /* Yes match found */
     }else{    /* Check for negative response */
          pattern = nl_langinfo(NOEXPR);
          if( ( status = regcomp( &re, pattern,
               REG_EXTENDED | REG_NOSUB )) != 0 ){
                    retval = -3;/*-3 indicates no compile error */
                    return(retval);
               }
          status = regexec( &re, string, (size_t)0,
                         (regmatch_t *)NULL, 0);
```

```
            if(status == 0)
                 retval = 0;/* Negative response match found */
        }else
                 retval = -1; /* The string did not match yes or no
                              response */
        regfree(&re);
        return(retval);
    }
```

Time Formatting Subroutines

In addition to the **strftime** subroutine defined in the C programming language standard, XPG4 defines the following time formatting subroutines:

wcsftime	Formats time into wide character code strings.
strptime	Converts a multibyte string into an internal time format.

Examples

1. The following example uses the **wcsftime** subroutine to format time into a wide character string:

```
#include <stdio.h>
#include <langinfo.h>
#include <locale.h>
#include <time.h>

main()
{
        wchar_t timebuf[BUFSIZE];
        time_t clock = time( (time_t*) NULL);
        struct tim *tmptr = localetime(&clock);

        (void)setlocale(LC_ALL, "");

        wcsftime(
                timebuf,       /* Time string output buffer */
                BUFSIZ,        /*Maximum size of output string */
                nl_langinfo(D_T_FMT),      /* Date/time format */
                tmptr          /* Pointer to tm structure */
        );

        printf("%S\n", timebuf);
}
```

2. The following example uses the **strptime** subroutine to convert a formatted time string to internal format:

```
#include <langinfo.h>
#include <locale.h>
#include <time.h>

main(int argc, char **argv)
{
        struct tm tm;

        (void)setlocale(LC_ALL, "");

        if (argc != 2) {
                ...                /* Error handling */
        }
        if (strptime(
                argv[1],           /* Formatted time string */
                nl_langinfo(D_T_FMT),      /* Date/time format */
                &tm                /* Address of tm structure */
        ) == NULL) {
```

```
                ...                 /* Error handling */
        }
        else {
                ...                /* Other Processing */
        }
}
```

Monetary Formatting Subroutines

Although the C programming language standard in conjunction with POSIX provides a means of specifying and accessing monetary information, these standards do not define a subroutine that formats monetary quantities. The XPG **strfmon** subroutine provides the facilities to format monetary quantities. There is no defined subroutine that converts a formatted monetary string into a numeric quantity suitable for arithmetic. Applications that need to do arithmetic on monetary quantities may do so after processing the locale-dependent monetary string into a number. The culture-specific monetary formatting information is specified by the **LC_MONETARY** category. An application can obtain information pertaining to the monetary format and the currency symbol by calling the **localeconv** subroutine.

Euro Currency Support via the @euro Modifier

The **strfmon** subroutine uses the information from the locale's **LC_MONETARY** category to determine the correct monetary format for the given language/territory. With the advent of the common European currency (Euro), locales must be able to handle both the traditional national currencies as well as the common European currency. This is accomplished via the @euro modifier. Each European country that uses the Euro will have an additional **LC_MONETARY** definition with the @euro modifier appended. This alternate format will be invoked when specified via the locale environment variables, or with the **setlocale** subroutine.

To use the French locale , UTF-8 codeset environment, and French francs as the monetary unit, simply set:

```
 LANG=FR_FR
```

To use the French locale, UTF-8 codeset environment,and Euros as the monetary unit, set:

```
 LANG=FR_FR
LC_MONETARY=FR_FR@euro
```

Users should NOT attempt to set LANG=FR_FR@euro, as the @euro variant for locale categories other than **LC_MONETARY** is undefined.

Examples

1. The following example uses the **strfmon** subroutine and accepts a format specification and an input value. The input value is formatted according to the input format specification.

```
#include <monetary.h>
#include <locale.h>
#include <stdio.h>

main(int argc, char **argv)
{
        char bfr[256], format[256];
        int match; ssize_t size;
        float value;

        (void) setlocal(LC_ALL, "");

        if (argc != 3){
                ...        /* Error handling */
        }
```

```
        match = sscanf(argv[1], "%f", &value);
        if (!match) {
                ...        /* Error handling */
        }
        match = sscanf(argv[2], "%s", format);
        if (!match) {
                ...        /*Error handling */
        }
        size = strfmon(bfr, 256, format, value);
        if (size == -1) {
                ...        /* Error handling */
        }
        printf ("Formatted monetary value is: %s\n", bfr);
}
```

The following table provides examples of some of the possible conversion specifications and the outputs for 12345.67 and -12345.67 in a US English locale:

Conversion Specification	Output	Description
%n	$12,345.67 -$12,345.67	Default formatting
%15n	$12,345.67 -$12,345.67	Right justifies within a 15-character field.
%#6n	$ 12,345.67 -$ 12,345.67	Aligns columns for values up to 999,999.
%=*#8n	$****12,345.67 -$****12,345.67	Specifies a fill character.
%=0#8n	$000012,345.67 -$000012,345.67	Fill characters do not use grouping.
%^#6n	$ 12345.67 -$ 12345.67	Disables the thousands separator.
%^#6.0n	$ 12346 -$ 12346	Rounds off to whole units.
%^#6.3n	$ 12345.670 -$ 12345.670	Increases the precision.
%(#6n	$ 12,345.67 ($ 12,345.67)	Uses an alternate positive or negative style.
%!(#6n	12,345.67 (12,345.67)	Disables the currency symbol.

2. The following example converts a monetary value into a numeric value. The monetary string is pointed to by `input` and the result of converting it into numeric form is stored in the string pointed to by `output`. Assume `input` and `output` are initialized.

```
char *input;   /* the input multibyte string containing the monetary string */
char *output;  /* the numeric string obtained from the input string */
wchar_t src_string[SIZE], dest_string[SIZE];
wchar_t *monetary, *numeric;
wchar_t mon_decimal_point, radixchar;
wchar_t wc;
localeconv *lc;

/* Initialize input and output to point to valid buffers as appropriate. */
/* Convert the input string to process code form*/
retval = mbstowcs(src_string, input, SIZE);
/* Handle error returns */

monetary = src_string;
numeric = dest_string;
lc = localeconv();
        /* obtain the LC_MONETARY and LC_NUMERIC info */

/* Convert the monetary decimal point to wide char form */
retval = mbtowc( &mon_decimal_point, lc->mon_decimal_point,
                MB_CUR_MAX);
/* Handle any error case */
```

```
/* Convert the numeric decimal point to wide char form */
retval = mbtowc( &radixchar, lc->decimal_point, MB_CUR_MAX);
/* Handle error case */
/* Assuming the string is converted first into wide character
** code form via mbstowcs, monetary points to this string.
*/
/* Pick up the numeric information from the wide character
** string and copy it into a temp buffer.
*/
        while(wc = *monetary++){
                if(iswdigit(wc))
                        *numeric++ = wc;
                else if( wc == mon_decimal_point)
                        *numeric++=radixchar;
        }
        *numeric = 0;
/* dest_string has the numeric value of the monetary quantity. */
/* Convert the numeric quantity into multibyte form */
retval = wcstombs( output, dest_string, SIZE);
/* Handle any error returns */
/* Output contains a numeric value suitable for atof conversion. */
```

Related Information

“National Language Support Subroutines Overview” on page 339 provides information about wide character and multibyte subroutines.

For general information about internationalizing programs, see National Language Support Overview for Programming (“Chapter 16. National Language Support” on page 329) and “Locale Overview for Programming” on page 330 .

The **strfmon** subroutine.

Multibyte and Wide Character Subroutines

This section contains information about multibyte and wide character code subroutines. This section contains the following major subsections:

Multibyte Code and Wide Character Code Conversion Subroutines

The internationalized environment of National Language Support blends multibyte and wide character subroutines. The decision of when to use wide character or multibyte subroutines can be made only after careful analysis.

If a program primarily uses multibyte subroutines, it may be necessary to convert the multibyte character codes to wide character codes before certain wide character subroutines can be used. If a program uses wide character subroutines, data may need to be converted to multibyte form when invoking subroutines.

Both methods have drawbacks, depending on the program in use and the availability of standard subroutines to perform the required processing. For instance, the wide character display-column-width subroutine has no corresponding standard multibyte subroutine.

If a program can process its characters in multibyte form, this method should be used instead of converting the characters to wide character form.

> **Attention**: The conversion between multibyte and wide character code depends on the current locale setting. Do not exchange wide character codes between two processes, unless you have knowledge that each locale that might used handles wide character codes in a consistent fashion. Most AIX locales use the Unicode character value as a wide character code, except locales based on the IBM-850 and IBM-eucTW codesets.

Multibyte Code to Wide Character Code Conversion Subroutines

The following subroutines are used when converting from multibyte code to wide character code:

mblen	Determines the length of a multibyte character.
mbstowcs	Converts a multibyte string to a wide character string.
mbtowc	Converts a multibyte character to a wide character.

Wide Character Code to Multibyte Code Conversion Subroutines

The following subroutines are used when converting from wide character code to multibyte character code:

wcslen	Determines the number of wide characters in a wide character string.
wcstombs	Converts a wide character string to a multibyte character string.
wctomb	Converts a wide character to a multibyte character.

Examples

1. The following example uses the **mbtowc** subroutine to convert a character in multibyte character code to wide character code:

```
main()
{
    char     *s;
    wchar_t  wc;
    int      n;

    (void)setlocale(LC_ALL,"");

    /*
    **  s points to the character string that needs to be
    **  converted to a wide character to be stored in wc.
    */
    n = mbtowc(&wc, s, MB_CUR_MAX);

    if (n == -1){
        /*  Error handle  */
    }
    if (n == 0){
        /*  case of name pointing to null  */
    }

    /*
    **  wc contains the process code for the multibyte character
    **  pointed to by s.
    */
}
```

2. The following example uses the **wctomb** subroutine to convert a character in wide character code to multibyte character code:

```
#include <stdlib.h>
#include <limits.h>             /* for MB_LEN_MAX  */
#include <stdlib.h>             /* for wchar_t     */

main()
{
    char    s[MB_LEN_MAX};      /*  system wide maximum number of
                                **  bytes in a multibyte character r. */
    wchar_t wc;
    int     n;

    (void)setlocale(LC_ALL,"");

    /*
    **  wc is the wide character code to be converted to
    **  multibyte character code.
    */
    n = wctomb(s, wc);

    if(n == -1){
        /* pwcs does not point to a valid wide character */
    }
    /*
    ** n has the number of bytes contained in the multibyte
    ** character stored in s.
    */
}
```

3. The following example uses the **mblen** subroutine to find the byte length of a character in multibyte character code:

```
#include <stdlib.h>
#include <locale.h>

main
{
    char *name = "h";
    int  n;

    (void)setlocale(LC_ALL,"");

    n = mblen(name, MB_CUR_MAX);
    /*
    **  The count returned in n is the multibyte length.
    **  It is always less than or equal to the value of
    **  MB_CUR_MAX in stdlib.h
    */
    if(n == -1){
        /* Error Handling */
    }
}
```

4. The following example obtains a previous character position in a multibyte string. If you need to determine the previous character position, starting from a current character position (not just some random byte position), step through the buffer starting at the beginning. Use the **mblen** subroutine until the current character position is reached and save the previous character position to obtain the needed character position.

```
char buf[];     /* contains the multibyte string */
char *cur,      /* points to the current character position */
char *prev,     /* points to previous multibyte character */
char *p;        /* moving pointer */

/* initialize the buffer and pointers as needed */
/* loop through the buffer until the moving pointer reaches
** the current character position in the buffer, always
```

```
** saving the last character position in prev pointer */
p = prev = buf;

/* cur points to a valid character somewhere in buf */
while(p< cur){
        prev = p;
        if( (i=mblen(p, mbcurmax))<=0){
                /* invalid multibyte character or null */
                /* You can have a different error handling
                ** strategy */
                p++;    /* skip it */
        }else {
                p += i;
        }
}
/* prev will point to the previous character position */

/* Note that if( prev == cur), then it means that there was
** no previous character. Also, if all bytes up to the
** current character are invalid, it will treat them as
** all valid single-byte characters and this may not be what
** you want. One may change this to handle another method of
** error recovery. */
```

5. The following example uses of the **mbstowcs** subroutine to convert a multibyte string to wide character string:

```
#include <stdlib.h>
#include <locale.h>

main()
{
     char    *s;
     wchar_t *pwcs;
     size_t  retval, n;

     (void)setlocale(LC_ALL, "");

     n = strlen(s) + 1;          /*string length + terminating null */

     /*  Allocate required wchar array    */
     pwcs = (wchar_t *)malloc(n * sizeof(wchar_t) );
     retval = mbstowcs(pwcs, s, n);
     if(retval == -1){

     /*  Error handle  */
          }
          /*
          ** pwcs contains the wide character string.
          */
}
```

6. The following example illustrates the problems with using the **mbstowcs** subroutine on a large block of data for conversion to wide character form. When it encounters an invalid multibyte, the **mbstowcs** subroutine returns a value of -1 but does not specify where the error occurred. Therefore, the **mbtowc** subroutine must be used repeatedly to convert one character at a time to wide character code.

   Note: Processing in this manner will considerably slow down program performance.

   During the conversion of single-byte code sets, there is no possibility for partial multibytes. However, during the conversion of multibyte code sets, partial multibytes are copied to a save buffer. During the next call to the **read** subroutine, the partial multibyte is prefixed to the rest of the byte sequence.

   Note: A null-terminated wide character string is obtained. Optional error handling can be done if an instance of an invalid byte sequence is found.

```
#include <stdio.h>
#include <locale.h>
#include <stdlib.h>

main(int argc, char *argv[])
{

    char     *curp, *cure;
    int      bytesread, bytestoconvert, leftover;
    int      invalid_multibyte, mbcnt, wcnt;
    wchar_t  *pwcs;
    wchar_t  wbuf[BUFSIZ+1];
    char     buf[BUFSIZ+1];
    char     savebuf[MB_LEN_MAX];
    size_t   mb_cur_max;
    int      fd;
        /*
        **  MB_LEN_MAX specifies the system wide constant for
        **  the maximum number of bytes in a multibyte character.
        */

    (void)setlocale(LC_ALL, "");
    mb_cur_max = MB_CUR_MAX;

    fd = open(argv[1], 0);
    if(fd < 0){
       /*  error handle  */
    }

    leftover = 0;
    if(mb_cur_max==1){    /*  Single byte code sets case  */
        for(;;){
            bytesread = read(fd, buf, BUSIZ);
            if(bytesread <= 0)
                break;
            mbstowcs(wbuf, buf, bytesread+1);
            /*  Process using the wide character buffer   */
        }
                        /*  File processed ...   */
        exit(0);        /*  End of program       */

    }else{              /*  Multibyte code sets  */
        leftover = 0;

        for(;;) {
            if(leftover)
                strncpy(buf, savebuf ,leftover);
            bytesread=read(fd,buf+leftover, BUFSIZ-leftover);
            if(bytesread <= 0)
                break;

            buf[leftover+bytesread] = '\0';
                     /* Null terminate string */
            invalid_multibyte = 0;
            bytestoconvert = leftover+bytesread;
            cure= buf+bytestoconvert;
            leftover=0;
            pwcs = wbuf;
                /* Stop processing when invalid mbyte found. */
            curp= buf;

            for(;curp<cure;){
                mbcnt = mbtowc(pwcs,curp, mb_cur_max);
                if(mbcnt>0){
                    curp += mbcnt;
                    pwcs++;
                    continue;
```

```
                }else{
                    /* More data needed on next read*/
                    if ( cure-curp<mb_cur_max){
                        leftover=cure-curp;
                        strncpy(savebuf,curp,leftover);
                        /* Null terminate before partial mbyte */
                        *curp=0;
                        break;

                    }else{
                            /*Invalid multibyte found */
                        invalid_multibyte =1;
                        break;
                    }
                }
            }
            if(invalid_multibyte){          /*error handle */
            }
            /* Process the wide char buffer */
        }
    }
}
```

7. The following example uses the **wcstombs** and **wcslen** subroutines to convert a wide character string to multibyte form:

```
#include <stdlib.h>
#include <locale.h>

main()
{
    wchar_t *pwcs; /* Source wide character string */
    char *s;       /* Destination multibyte character string */
    size_t n;
    size_t retval;

    (void)setlocale(LC_ALL, "");
    /*
    ** Calculate the maximum number of bytes needed to
    ** store the wide character buffer in multibyte form in the
    ** current code page and malloc() the appropriate storage,
    ** including the terminating null.
    */
    s = (char *) malloc( wcslen(pwcs) * MB_CUR_MAX + 1 );
    retval= wcstombs( s, pwcs, n);
    if( retval == -1) {
        /* Error handle */
        /* s points to the multibyte character string. */
}
```

Wide Character Classification Subroutines

The majority of wide character classification subroutines are similar to traditional character classification subroutines, except that wide character classification subroutines operate on a **wchar_t** data type argument passed as a **wint_t** data type argument.

Generic Wide Character Classification Subroutines

In the internationalized environment of National Language Support, the ability to create new character class properties is essential. For example, several properties are defined for Japanese characters that are not applicable to the English language. As more languages are supported, a framework enabling applications to deal with a varying number of character properties is needed. The **wctype** and **iswctype** subroutines allow handling of character classes in a general fashion. These subroutines are used to allow for both user-defined and language-specific character classes.

The action of wide character classification subroutines is affected by the definitions in the **LC_CTYPE** category for the current locale.

To create new character classifications for use with the **wctype** and **iswctype** subroutines, create a new character class in the **LC_CTYPE** category and generate the locale using the **localedef** command. A user application obtains this locale data with the **setlocale** subroutine. The program can then access the new classification subroutines by using the **wctype** subroutine to get the **wctype_t** property handle. It then passes to the **iswctype** subroutine both the property handle and the wide character code of the character to be tested.

wctype Obtains handle for character property classification.
iswctype Tests for character property.

Standard Wide Character Classification Subroutines

The **isw*** subroutines determine various aspects of a standard wide character classification. The **isw*** subroutines also work with single-byte code sets. The **isw*** subroutines should be used in preference to the **wctype** and **iswctype** subroutines. The **wctype** and **iswctype** subroutines should be used only for extended character class properties (for example, Japanese language properties).

When using the wide character functions to convert the case in several blocks of data, the application must convert characters from multibyte to wide character code form. Since this may affect performance in single-byte code set locales, you should consider providing two conversion paths in your application. The traditional path for single-byte code set locales would convert case using the **isupper**,**islower**, **toupper**, and **tolower** subroutines. The alternate path for multibyte code set locales would convert multibyte characters to wide character code form and convert case using the **iswupper**, **iswlower**, **towupper** and **towlower** subroutines. When converting multibyte characters to wide character code form, an application needs to handle special cases where a multibyte character may split across successive blocks.

iswalnum Tests for alphanumeric character classification.
iswalpha Tests for alphabetic character classification.
iswcntrl Tests for control character classification.
iswdigit Tests for digit character classification.
iswgraph Tests for graphic character classification.
iswlower Tests for lowercase character classification.
iswprint Tests for printable character classification.
iswpunct Tests for punctuation character classification.
iswspace Tests for space character classification.
iswupper Tests for uppercase character classification.
iswxdigit Tests for hexadecimal-digit character classification.

Wide Character Case Conversion Subroutines

The following subroutines convert cases for wide characters. The action of wide character case conversion subroutines is affected by the definition in the **LC_CTYPE** category for the current locale.

towlower Converts an uppercase wide character to a lowercase wide character.
towupper Converts a lowercase wide character to an uppercase wide character.

Example

The following example uses the **wctype** subroutine to test for the **NEW_CLASS** character classification:

```
#include <ctype.h>
#include <locale.h>
#include <stdlib.h>
```

```
main()
{
    wint_t    wc;
    int       retval;
    wctype_t  chandle;

    (void)setlocale(LC_ALL,"");
    /*
    ** Obtain the character property handle for the NEW_CLASS
    ** property.
    */
    chandle = wctype("NEW_CLASS") ;
    if(chandle == (wctype_t)0){
        /* Invalid property. Error handle. */
    }
    /* Let wc be the wide character code for a character */
    /* Test if wc has the property of NEW_CLASS */
    retval = iswctype( wc, chandle );
    if( retval > 0 ) {
        /*
        ** wc has the property NEW_CLASS.
        */
    }else if(retval == 0) {
        /*
        ** The character represented by wc does not have the
        ** property NEW_CLASS.
        */
    }
}
```

Wide Character Display Column Width Subroutines

When characters are displayed or printed, the number of columns occupied by a character may differ. For example, a Kanji character (Japanese language) may occupy more than one column position. The number of display columns required by each character is part of the National Language Support locale database. The **LC_CTYPE** category defines the number of columns needed to display a character.

There are no standard multibyte display-column-width subroutines. For portability, convert multibyte codes to wide character codes and use the required wide character display-width subroutines. However, if the **__max_disp_width** macro (defined in the **stdlib.h** file) is set to 1 and a single-byte code set is in use, then the display-column widths of all characters (except tabs) in the code set are the same, and are equal to 1. In this case, the **strlen (***string***)** subroutine gives the display column width of the specified string. This is demonstrated in the following example:

```
#include <stdlib.h>
        int display_column_width;  /* number  of  display  columns  */
        char  *s;                  /*  character  string            */
        ....
        if((MB_CUR_MAX  ==  1)  &&  (__max_disp_width  ==  1)){
                display_column_width  =  strlen(s);
                                   /*  s  is  a  string  pointer    */
        }
```

The following subroutines find the display widths for wide character strings:

wcswidth Determines the display width of a wide character string.
wcwidth Determines the display width of a wide character.

Examples

1. The following example uses the **wcwidth** subroutine to find the display column width of a wide character:

```
#include  <string.h>
#include  <locale.h>
#include  <stdlib.h>

main()
{
    wint_t  wc;
    int     retval;

    (void)setlocale(LC_ALL,  "");

    /*
    **    Let wc be the wide character whose display width is
    **    to be found.
    */
    retval  =  wcwidth(wc);
    if(retval  ==  -1){
        /*
        **  Error handling. Invalid or nonprintable
        **  wide character in wc.
        */
    }
}
```

2. The following example uses the **wcswidth** subroutine to find the display column width of a wide character string:

```
#include  <string.h>
#include  <locale.h>
#include  <stdlib.h>

main()
{
    wchar_t  *pwcs;
    int      retval;
    size_t   n;

    (void)setlocale(LC_ALL,  "");
    /*
    **    Let pwcs point to a wide character null
    **    terminated string.
    **    Let n be the number of wide characters
    **    whose display column width is to be determined.
    */
    retval  =  wcswidth(pwcs,  n);
    if(retval  ==  -1){
        /*
        **  Error handling. Invalid wide or nonprintable
        **  character  ode encountered in the wide
        **  character string pwcs.
        */
    }
}
```

Multibyte and Wide Character String Collation Subroutines

Strings can be compared in two ways:

- Using the ordinal (binary) values of the characters.
- Using the weights associated with the characters for each locale, as determined by the **LC_COLLATE** category.

National Language Support (NLS) uses the second method.

Collation is a locale-specific property of characters. A weight is assigned to each character to indicate its relative order for sorting. A character may be assigned more than one weight. Weights are prioritized as primary, secondary, tertiary, and so forth. The maximum number of weights assigned each character is system-defined.

A process inherits the C locale or POSIX locale at its startup time. When the **setlocale (LC_ALL, ″ ″)** subroutine is called, a process obtains its locale based on the **LC_*** and **LANG** environment variables. The following subroutines are affected by the **LC_COLLATE** category and determine how two strings will be sorted in any given locale.

> **Note:** Collation-based string comparisons take a long time because of the processing involved in obtaining the collation values. Such comparisons should be used only when necessary. If you need to find whether two wide character strings are equal, do not use the **wcscoll** and **wcsxfrm** subroutines. Use the **wcscmp** subroutine instead.

The following subroutines compare multibyte character strings:

strcoll	Compares the collation weights of multibyte character strings.
strxfrm	Converts a multibyte character string to values representing character collation weights.

The following subroutines compare wide character strings:

wcscoll	Compares the collation weights of wide character strings.
wcsxfrm	Converts a wide character string to values representing character collation weights.

Examples

1. The following example uses the **wcscoll** subroutine to compare two wide character strings based on their collation weights:

```
#include  <stdio.h>
#include  <string.h>
#include  <locale.h>
#include  <stdlib.h>

extern  int  errno;

main()
{
    wchar_t  *pwcs1, *pwcs2;
    size_t   n;

    (void)setlocale(LC_ALL,  "");

    /*    set it to zero for checking errors on wcscoll    */
    errno  =  0;
    /*
    **    Let pwcs1 and pwcs2 be two wide character strings to
    **    compare.
    */
    n  =  wcscoll(pwcs1, pwcs2);
        /*
        **    If errno is set then it indicates some
        **    collation error.
        */
    if(errno  !=  0){
        /*  error has occurred... handle error ...*/
    }
}
```

2. The following example uses the **wcsxfrm** subroutine to compare two wide character strings based on collation weights:

Note: Determining the size *n* (where *n* is a number) of the transformed string, when using the **wcsxfrm** subroutine, can be accomplished in one of the following ways:

a. For each character in the wide character string, the number of bytes for possible collation values cannot exceed the **COLL_WEIGHTS_MAX * sizeof(wchar_t)** value. This value, multiplied by the number of wide character codes, gives the buffer length needed. To the buffer length add 1 for the terminating wide character null. This strategy may slow down performance.

b. Estimate the byte-length needed. If the previously obtained value is not enough, increase it. This may not satisfy all strings but gives maximum performance.

c. Call the **wcsxfrm** subroutine twice: once to find the value of *n*, and again to transform the string using this *n* value. This strategy slows down performance because the **wcsxfrm** subroutine is called twice. However, it yields a precise value for the buffer size needed to store the transformed string.

Which method to choose depends on the characteristics of the strings used in the program and the performance objectives of the program.

```
#include  <stdio.h>
#include  <string.h>
#include  <locale.h>
#include  <stdlib.h>

main()
{
    wchar_t  *pwcs1, *pwcs2, *pwcs3, *pwcs4;
    size_t   n, retval;

    (void)setlocale(LC_ALL, "");
    /*
    **  Let the string pointed to by pwcs1 and pwcs3 be the
    **  wide character arrays to store the transformed wide
    **  character strings. Let the strings pointed to by pwcs2
    **  and pwcs4 be the wide character strings to compare based
    **  on the collation values of the wide characters in these
    **  strings.
    **  Let n be large enough (say,BUFSIZ) to transform the two
    **  wide character strings specified by pwcs2 and pwcs4.
    **
    **  Note:
    **  In practice, it is best to call wcsxfrm if the wide
    **  character string is to be compared several times to
    **  different wide character strings.
    */

    do {
        retval = wcsxfrm(pwcs1, pwcs2, n);
        if(retval == (size_t)-1){
            /*  error has occurred.  */
            /*  Process the error if needed  */
            break;
        }

        if(retval >= n ){
        /*
        ** Increase the value of n and use a bigger buffer pwcs1.
        */
        }
    }while (retval >= n);

    do {
        retval = wcsxfrm(pwcs3, pwcs4, n);
        if (retval == (size_t)-1){
            /*  error has occurred.  */
            /*  Process the error if needed  */
            break;
```

```
            if(retval >= n){
            /*Increase the value of n and use a bigger buffer pwcs3.*/
            }
        }while (retval >= n);
        retval = wcscmp(pwcs1, pwcs3);
        /*  retval has the result  */
    }
```

Multibyte and Wide Character String Comparison Subroutines

The **strcmp** and **strncmp** subroutines determine if the contents of two multibyte strings are equivalent. If your application needs to know how the two strings differ lexically, use the multibyte and wide character string collation subroutines.

The following NLS subroutines compare wide character strings:

wcscmp	Compares two wide character strings.
wcsncmp	Compares a specific number of wide character strings.

Example

The following example uses the **wcscmp** subroutine to compare two wide character strings:

```
#include <string.h>
#include <locale.h>
#include <stdlib.h>

main()
{
    wchar_t *pwcs1, *pwcs2;
    int retval;

    (void)setlocale(LC_ALL, "");
    /*
    **  pwcs1 and pwcs2 point to two wide character
    **  strings to compare.
    */
    retval = wcscmp(pwcs1, pwcs2);
    /*  pwcs1 contains a copy of the wide character string
    **  in pwcs2
    */
}
```

Wide Character String Conversion Subroutines

The following NLS subroutines convert wide character strings to double, long, and unsigned long integers:

wcstod	Converts a wide character string to a double-precision floating point.
wcstol	Converts a wide character string to a signed long integer.
wcstoul	Converts a wide character string to an unsigned long integer.

Before calling the **wcstod**, **wcstoul**, or **wcstol** subroutine, the **errno** global variable must be set to 0. Any error that occurs as a result of calling these subroutines can then be handled correctly.

Examples

1. The following example uses the **wcstod** subroutine to convert a wide character string to a double-precision floating point:

   ```
   #include <stdlib.h>
   #include <locale.h>
   #include <errno.h>
   ```

```
extern int errno;

main()
{
    wchar_t  *pwcs, *endptr;
    double   retval;

    (void)setlocale(LC_ALL, "");
    /*
    **  Let pwcs point to a wide character null terminated
    **  string containing a floating point value.
    */
    errno = 0;   /*  set errno to  zero  */
    retval = wcstod(pwcs, &endptr);

    if(errno != 0){
        /*  errno has changed, so error has occurred */

        if(errno == ERANGE){
            /*  correct value is outside range of
            **  representable values. Case of overflow
            **  error
            */

             if((retval == HUGE_VAL) ||
                (retval == -HUGE_VAL)){
                /*  Error case. Handle accordingly.  */
            }else if(retval == 0){
                /*  correct value causes underflow   */
                /*  Handle appropriately             */
            }
        }
    }
    /*  retval contains the double.  */
}
```

2. The following example uses the **wcstol** subroutine to convert a wide character string to a signed long integer:

```
#include <stdlib.h>
#include <locale.h>
#include <errno.h>
#include <stdio.h>

extern int errno;

main()
{
    wchar_t   *pwcs, *endptr;
    long int  retval;

    (void)setlocale(LC_ALL, "");
    /*
    **  Let pwcs point to a wide character null terminated
    **  string containing a signed long integer value.
    */
    errno = 0;  /*  set errno to  zero  */
    retval = wcstol(pwcs, &endptr, 0);

    if(errno != 0){
        /*  errno has changed, so error has occurred */

        if(errno == ERANGE){
            /*  correct value is outside range of
            **  representable values. Case of overflow
            **  error
            */
```

```
            if((retval == LONG_MAX) || (retval == LONG_MIN)){
               /*  Error case. Handle accordingly.     */
            }else if(errno == EINVAL){
               /*  The value of base is not supported  */
               /*  Handle appropriately                */
            }
        }
    }
    /*  retval contains the long integer.  */
}
```

3. The following example uses the **wcstoul** subroutine to convert a wide character string to an unsigned long integer:

```
#include <stdlib.h>
#include <locale.h>
#include <errno.h>

extern int errno;

main()
{
    wchar_t    *pwcs, *endptr;
    unsigned long int  retval;

    (void)setlocale(LC_ALL, "");

    /*
    **  Let pwcs point to a wide character null terminated
    **  string containing an unsigned long integer value.
    */
    errno = 0;   /*  set errno to  zero  */
    retval = wcstoul(pwcs, &endptr, 0);

    if(errno != 0){
        /*  error has occurred */
        if(retval == ULONG_MAX || errno == ERANGE){
            /*
            **  Correct value is outside of
            **  representable value. Handle appropriately
            */
        }else if(errno == EINVAL){
            /*  The value of base is not representable  */
            /*  Handle appropriately                    */
        }
    }
    /*  retval contains the unsigned long integer.  */
}
```

Wide Character String Copy Subroutines

The following NLS subroutines copy wide character strings:

wcscpy	Copies a wide character string to another wide character string.
wcsncpy	Copies a specific number of characters from a wide character string to another wide character string.
wcscat	Appends a wide character string to another wide character string.
wcsncat	Appends a specific number of characters from a wide character string to another wide character string.

Example

The following example uses the **wcscpy** subroutine to copy a wide character string into a wide character array:

```
#include <string.h>
#include <locale.h>
#include <stdlib.h>

main()
{
    wchar_t *pwcs1, *pwcs2;
    size_t  n;

    (void)setlocale(LC_ALL, "");
    /*
    **  Allocate the required wide character array.
    */
    pwcs1 = (wchar_t *)malloc( (wcslen(pwcs2) +1)*sizeof(wchar_t));
    wcscpy(pwcs1, pwcs2);
    /*
    **  pwcs1 contains a copy of the wide character string in pwcs2
    */
}
```

Wide Character String Search Subroutines

The following NLS subroutines are used to search for wide character strings:

wcschr	Searches for the first occurrence of a wide character in a wide character string.
wcsrchr	Searches for the last occurrence of a wide character in a wide character string.
wcspbrk	Searches for the first occurrence of a several wide characters in a wide character string.
wcsspn	Determines the number of wide characters in the initial segment of a wide character string.
wcscspn	Searches for a wide character string.
wcswcs	Searches for the first occurrence of a wide character string within another wide character string.
wcstok	Breaks a wide character string into a sequence of separate wide character strings.

Examples

1. The following example uses the **wcschr** subroutine to locate the first occurrence of a wide character in a wide character string:

```
#include <string.h>
#include <locale.h>
#include <stdlib.h>

main()
{
    wchar_t *pwcs1, wc, *pws;
    int     retval;

    (void)setlocale(LC_ALL, "");

    /*
    **  Let pwcs1 point to a wide character null terminated string.
    **  Let wc point to the wide character to search for.
    **
    */
    pws = wcschr(pwcs1, wc);
    if (pws == (wchar_t )NULL ){
        /*  wc does not occur in pwcs1  */
    }else{
        /*  pws points to the location where wc is found  */
    }
}
```

2. The following example uses the **wcsrchr** subroutine to locate the last occurrence of a wide character in a wide character string:

```
#include <string.h>
#include <locale.h>
#include <stdlib.h>

main()
{
    wchar_t *pwcs1, wc, *pws;
    int     retval;

    (void)setlocale(LC_ALL, "");
    /*
    **  Let pwcs1 point to a wide character null terminated string.
    **  Let wc point to the wide character to search for.
    **
    */
    pws = wcsrchr(pwcs1, wc);
    if (pws == (wchar_t )NULL ){
        /*  wc does not occur in pwcs1  */
    }else{
        /*  pws points to the location where wc is found  */
    }
}
```

3. The following example uses the **wcspbrk** subroutine to locate the first occurrence of several wide characters in a wide character string:

```
#include <string.h>
#include <locale.h>
#include <stdlib.h>

main()
{
    wchar_t *pwcs1, *pwcs2, *pws;

    (void)setlocale(LC_ALL, "");

    /*
    **  Let pwcs1 point to a wide character null terminated string.
    **  Let pwcs2 be initialized to the wide character string
    **  that contains wide characters to search for.
    */
    pws = wcspbrk(pwcs1, pwcs2);

    if (pws == (wchar_t )NULL ){
        /* No wide character from pwcs2 is found in pwcs1    */
    }else{
        /* pws points to the location where a match is found */
    }
}
```

4. The following example uses the **wcsspn** subroutine to determine the number of wide characters in the initial segment of a wide character string segment:

```
#include <string.h>
#include <locale.h>
#include <stdlib.h>

main()
{
    wchar_t *pwcs1, *pwcs2;
    size_t count;

    (void)setlocale(LC_ALL, "");
    /*
    **  Let pwcs1 point to a wide character null terminated string.
    **  Let pwcs2 be initialized to the wide character string
    **  that contains wide characters to search for.
    */
    count = wcsspn(pwcs1, pwcs2);
```

```
    /*
    **  count contains the length of the segment.
    */
 }
```

5. The following example uses the **wcscspn** subroutine to determine the number of wide characters not in a wide character string segment:

```
#include <string.h>
#include <locale.h>
#include <stdlib.h>

main()
{
    wchar_t *pwcs1, *pwcs2;
    size_t count;

    (void)setlocale(LC_ALL, "");

    /*
    **  Let pwcs1 point to a wide character null terminated string.
    **  Let pwcs2 be initialized to the wide character string
    **  that contains wide characters to search for.
    */
    count = wcscspn(pwcs1, pwcs2);
    /*
    **  count contains the length of the segment consisting
    **  of characters not in pwcs2.
    */
}
```

6. The following example uses the **wcswcs** subroutine to locate the first occurrence of a wide character string within another wide character string:

```
#include <string.h>
#include <locale.h>
#include <stdlib.h>

main()
{
    wchar_t *pwcs1, *pwcs2, *pws;

    (void)setlocale(LC_ALL, "");
    /*
    **  Let pwcs1 point to a wide character null terminated string.
    **  Let pwcs2 be initialized to the wide character string
    **  that contains wide characters sequence to locate.
    */
    pws = wcswcs(pwcs1, pwcs2);
    if (pws == (wchar_t)NULL){
       /*  wide character sequence pwcs2 is not found in pwcs1 */
    }else{
        /*
        **  pws points to the first occurrence of the sequence
        **  specified by pwcs2 in pwcs1.
        */
    }
}
```

7. The following example uses the **wcstok** subroutine to tokenize a wide character string:

```
#include <string.h>
#include <locale.h>
#include <stdlib.h>

main()
{
    wchar_t *pwcs1 = L"?a???b,,,#c";
    wchar_t *pwcs;
```

```
    (void)setlocale(LC_ALL, "");
    pwcs = wcstok(pwcs1, L"?");
    /*  pws points to the token:  L"a"  */
    pwcs = wcstok((wchar_t *)NULL, L",");
    /*  pws points to the token:  L"??b"  */
    pwcs = wcstok((wchar_t *)NULL, L"#,");
    /*  pws points to the token:  L"c"  */
}
```

Wide Character Input/Output Subroutines

NLS provides subroutines for both formatted and unformatted I/O.

Formatted Wide Character I/O

Additions to the **printf** and **scanf** family of subroutines allow for the formatting of wide characters. The **printf** and **scanf** subroutines have two additional format specifiers for wide character handling: %C and %S. The %C and %S format specifiers allow I/O on a wide character and a wide character string, respectively. They are similar to the %c and %s format specifiers, which allow I/O on a multibyte character and string.

The multibyte subroutines accept a multibyte array and output a multibyte array. To convert multibyte output from a multibyte subroutine to a wide character string, use the **mbstowcs** subroutine.

Unformatted Wide Character I/O

Unformatted wide character I/O subroutines are used when a program requires code set-independent I/O for characters from multibyte code sets. For example, the **fgetwc** or **getwc** subroutine should be used to input a multibyte character. If the program uses the **getc** subroutine to input a multibyte character, the program must call the **getc** subroutine once for each byte in the multibyte character.

Wide character input subroutines read multibyte characters from a stream and convert them to wide characters. The conversion is done as if the subroutines call the **mbtowc** and **mbstowcs** subroutines.

Wide character output subroutines convert wide characters to multibyte characters and write the result to the stream. The conversion is done as if the subroutines call the **wctomb** and **wcstombs** subroutines.

The behavior of wide character I/O subroutines is affected by the **LC_CTYPE** category of the current locale.

Reading and Processing an Entire File: If a program has to go through an entire file that must be handled in wide character code form, it can be done in one of the following ways:

- In the case of multibyte characters, use either the **read** or **fread** subroutine to convert a block of text data into a buffer. Convert one character at a time in this buffer using the **mbtowc** subroutine. Handle special cases of multibyte characters crossing block boundaries. For multibyte code sets, do not use the **mbstowcs** subroutine on this buffer. On an invalid or a partial multibyte character sequence, the **mbstowcs** subroutine returns -1 without indicating how far it successfully converted the data. You can use the **mbstowcs** subroutine with single-byte code sets because you will not run into a partial-byte sequence problem with single-byte code sets.
- Use the **fgetws** subroutine to obtain a line from the file. If the returned wide character string contains a wide character <new-line>, then a complete line is obtained. If there is no <new-line> wide character, it means that the line is longer than expected, and more calls to the **fgetws** subroutine are needed to obtain the complete line. If the program can efficiently process one line at a time, this approach is recommended.
- If the **fgets** subroutine is used to read a multibyte file to obtain one line at a time, a split multibyte character may result. This condition needs to be handled just as in the case of the **read** subroutine breaking up a multibyte character across successive reads. If you can guarantee that the input line

length is not more than a set limit, a buffer of that size (plus 1 for null) can be used, thereby avoiding the possibility of a split multibyte character. If the program can efficiently process one line at a time, this approach may be used. Because of the possibility of split bytes in the buffer, you should use the **fgetws** subroutine in preference to the **fgets** subroutine for multibyte characters.

- Use the **fgetwc** subroutine on the file to read one wide character code at a time. If a file is large, the function call overhead becomes large and reduces the value of this method.

The decision of which one of these methods to use should be made on a per program basis. The second option is recommended, as it is capable of high performance and the program does not have to handle the special cases.

Input Subroutines: A new data type, **wint_t**,is required to represent the wide character code value as well as the end-of-file (EOF) marker. For example, consider the case of the **fgetwc** subroutine, which returns a wide character code value:

wchar_t fgetwc();	If the **wchar_t** data type is defined as a **char** value, the y-umlaut symbol cannot be distinguished from the end-of-file (EOF) marker in the ISO8859-1 code set. The 0xFF code point is a valid character (y umlaut). Hence, the return value cannot be the **wchar_t** data type. A data type is needed that can hold both the EOF marker and all the code points in a code set.
int fgetwc();	On some machines, the **int** data type is defined to be 16 bits. When the **wchar_t** data type is larger than 16 bits, the **int** value cannot represent all the return values.

Due to these reasons, **wint_t** data type is needed to represent the **fgetwc** subroutine return value. The **wint_t** data type is defined in the **wchar.h** file.

The following subroutines are used for wide character input:

fgetwc	Gets next wide character from a stream.
fgetws	Gets a string of wide characters from a stream.
getwc	Gets next wide character from a stream.
getwchar	Gets next wide character from standard input.
getws	Gets a string of wide characters from a standard input.
ungetwc	Pushes a wide character onto a stream.

Output Subroutines: The following subroutines are used for wide character output:

fputwc	Writes a wide character to an output stream.
fputws	Writes a wide character string to an output stream.
putwc	Writes a wide character to an output stream.
putwchar	Writes a wide character to standard output.
putws	Writes a wide character string to standard output.

Examples

1. The following example uses the **fgetwc** subroutine to read wide character codes from a file:

```
#include <stdio.h>
#include <locale.h>
#include <stdlib.h>

main()
{
    wint_t  retval;
    FILE    *fp;
    wchar_t *pwcs;

    (void)setlocale(LC_ALL, "");
```

```
    /*
    **  Open a stream.
    */
    fp = fopen("file", "r");

    /*
    **  Error Handling if fopen was not successful.
    */
    if(fp == NULL){
        /*  Error handler  */
    }else{
        /*
        **  pwcs points to a wide character buffer of BUFSIZ.
        */
        while((retval = fgetwc(fp)) != WEOF){
            *pwcs++ = (wchar_t)retval;
               /*  break when buffer is full  */
        }
    }
    /*  Process the wide characters in the buffer  */
}
```

2. The following example uses the **getwchar** subroutine to read wide characters from standard input:

```
#include <stdio.h>
#include <locale.h>
#include <stdlib.h>

 main()
{
    wint_t  retval;
    FILE    *fp;
    wchar_t *pwcs;

    (void)setlocale(LC_ALL, "");

    index = 0;
    while((retval = getwchar()) != WEOF){
        /*  pwcs points to a wide character buffer of BUFSIZ.  */
        *pwcs++ = (wchar_t)retval;
        /*  break on buffer full  */
    }
    /*  Process the wide characters in the buffer  */
}
```

3. The following example uses the **ungetwc** subroutine to push a wide character onto an input stream:

```
#include <stdio.h>
#include <locale.h>
#include <stdlib.h>

main()
{
    wint_t  retval;
    FILE    *fp;

    (void)setlocale(LC_ALL, "");
    /*
    **  Open a stream.
    */
    fp = fopen("file", "r");

    /*
    **  Error Handling if fopen was not successful.
    */
    if(fp == NULL){
        /*  Error handler  */
```

```
    else{
        retval = fgetwc(fp);
        if(retval != WEOF){
            /*
            **  Peek at the character and return it to the stream.
            */
            retval = ungetwc(retval, fp);
            if(retval == EOF){
                /*  Error on ungetwc  */
            }
        }
    }
}
```

4. The following example uses the **fgetws** subroutine to read a file one line at a time:

```
#include <stdio.h>
#include <locale.h>
#include <stdlib.h>

main()
{
    FILE    *fp;
    wchar_t *pwcs;

    (void)setlocale(LC_ALL, "");

    /*
    **  Open a stream.
    */
    fp = fopen("file", "r");

    /*
    **  Error Handling if fopen was not successful.
    */
    if(fp == NULL){
        /*  Error handler  */
    }else{
        /*  pwcs points to wide character buffer of BUFSIZ.  */
        while(fgetws(pwcs, BUFSIZ, fp) != (wchar_t *)NULL){
            /*
            **  pwcs contains wide characters with null
            **  termination.
            */
        }
    }
}
```

5. The following example uses the **fputwc** subroutine to write wide characters to an output stream:

```
#include <stdio.h>
#include <locale.h>
#include <stdlib.h>

main()
{
    int     index, len;
    wint_t  retval;
    FILE    *fp;
    wchar_t *pwcs;

    (void)setlocale(LC_ALL, "");

    /*
    **  Open a stream.
    */
    fp = fopen("file", "w");

    /*
```

```
    **  Error Handling if fopen was not successful.
    */
    if(fp == NULL){
        /*  Error handler  */
    }else{
        /*  Let len indicate number of wide chars to output.
        **  pwcs points to a wide character buffer of BUFSIZ.
        */
        for(index=0; index < len; index++){
            retval = fputwc(*pwcs++, fp);
            if(retval == WEOF)
                break;  /*  write error occurred  */
                        /*  errno is set to indicate the error. */
        }
    }
}
```

6. The following example uses the **fputws** subroutine to write a wide character string to a file:

```
#include <stdio.h>
#include <locale.h>
#include <stdlib.h>

main()
{
    int     retval;
    FILE    *fp;
    wchar_t *pwcs;

    (void)setlocale(LC_ALL, "");

    /*
    **  Open a stream.
    */
    fp = fopen("file", "w");

    /*
    **  Error Handling if fopen was not successful.
    */
    if(fp == NULL){
        /*  Error handler  */

    }else{
        /*
        **  pwcs points to a wide character string
        **  to output to fp.
        */
        retval = fputws(pwcs, fp);
        if(retval == -1){
            /*  Write error occurred                */
            /*  errno is set to indicate the error  */
        }
    }
}
```

Working with the Wide Character Constant

Use the **L** constant for ASCII characters only. For ASCII characters, the **L** constant value is numerically the same as the code point value of the character. For example, `L'a'` is same as `a`. The reason for using the **L** constant is to obtain the **wchar_t** value of an ASCII character for assignment purposes. A wide character constant is introduced by the **L** specifier. For example:

```
wchar_t wc = L'x' ;
```

A wide character code corresponding to the character `x` is stored in `wc`. The C compiler converts the character `x` using the **mbtowc** or **mbstowcs** subroutine as appropriate. This conversion to wide characters

is based on the current locale setting at compile time. Because ASCII characters are part of all supported code sets and the wide character representation of all ASCII characters is the same in all locales, L'x' results in the same value across all code sets. However, if the character x is non-ASCII, the program may not work when it is run on a different code set than used at compile time. This limitation impacts some programs that use switch statements using the wide character constant representation.

See the following partial program ″example″, compiled using the IBM-850 code set.

```
wchar_t
wc;
switch(wc){
     case L'a-umlaut':/*substitute the a-umlaut character here*/
               /*Process*/
          break;
     case :L'c-cedilla':/*substitute the c-cedilla character here*/
               /*Process*/
          break;
     default:
          break;
}
```

If this program is compiled and executed on an IBM-850 code set system, it will run correctly. However, if the same executable is run on an ISO8859-1 system, it may not work correctly. The characters *a-umlaut* and *c-cedilla* may have different process codes in IBM-850 and ISO8859-1 code sets.

Related Information

“National Language Support Subroutines Overview” on page 339 provides information about wide character and multibyte subroutines.

For general information about internationalizing programs, see “Chapter 16. National Language Support” on page 329 and “Locale Overview for Programming” on page 330.

The **LC_COLLATE** category of the locale definition file in *AIX 5L Version 5.1 Files Reference*.

The **LC_CTYPE** category of the locale definition file in *AIX 5L Version 5.1 Files Reference*.

The **localedef** command in *AIX 5L Version 5.1 Commands Reference, Volume 3*

“List of Wide Character Subroutines” on page 503 and “List of Multibyte Character Subroutines” on page 503

The **getc** subroutine, **printf** subroutines, in *AIX 5L Version 5.1 Technical Reference: Base Operating System and Extensions Volume 1*; and **read** subroutine, **scanf** subroutines, **setlocale** subroutine, **strlen** subroutine in *AIX 5L Version 5.1 Technical Reference: Base Operating System and Extensions Volume 2*.

Internationalized Regular Expression Subroutines

The following subroutines are available for use with internationalized regular expressions.

regcomp	Compiles a specified basic or extended regular expression into an executable string.
regexec	Compares a null-terminated string with a compiled basic or extended regular expression that must have been previously compiled by a call to the **regcomp** subroutine.
regerror	Provides a mapping from error codes returned by the **regcomp** and **regexec** subroutines to printable strings.
regfree	Frees any memory allocated by the **regcomp** subroutine associated with the compiled basic or extended regular expression. The expression is no longer treated as a compiled basic or extended regular expression after it is given to the **regfree** subroutine.

fnmatch Checks a specified string to see if it matches a specified pattern. You can use the **fnmatch** subroutine in an application that reads a dictionary to find which entries match a given pattern. You also can use the **fnmatch** subroutine to match pathnames to patterns.

Examples

1. The following example compiles an internationalized regular expression and matches a string using this compiled expression. A match is found for the first pattern, but no match is found for the second pattern.

```
#include                <locale.h>
#include                <regex.h>

#define                 BUFSIZE    256


main()
{
        char    *p;

        char    *pattern[] = {
                "hello[0-9]*",
                "1234"
        };


        char     *string = "this is a test string hello112 and this is test";
                /* This is the source string for matching */

        int     retval;
        regex_t   re;
        char    buf[BUFSIZE];

        int     i;

        setlocale(LC_ALL, "");

        for(i = 0;i <2; i++){
                retval = match(string, pattern[i], &re);
                if(retval == 0){
                        printf("Match found \n");
                }else{
                        regerror(retval, &re, buf, BUFSIZE);
                        printf("error = %s\n", buf);
                }
        }
        regfree( &re);
}


int match(char *string, char *pattern, regex_t *re)
{
        int     status;

        if((status=regcomp( re, pattern, REG_EXTENDED))!= 0)
                return(status);
        status = regexec( re, string, 0, NULL, 0);
        return(status);
}
```

2. The following example finds all substrings in a line that match a pattern. The numbers `11` and `1992` are matched. Every digit that is matched counts as one match. There are six such matches corresponding to the six digits supplied in the string.

```
#include         <locale.h>
#include                <regex.h>

#define                BUFSIZE   256


main()
{
        char    *p;

        char    *pattern = "[0-9]";
        char    *string = "Today is 11 Feb 1992 ";

        int     retval;
        regex_t re;
        char    buf[BUFSIZE];
        regmatch_t pmatch[100];
        int      status;
        char     *ps;

        int      eflag;

        setlocale(LC_ALL, "");

        /* Compile the pattern */

        if((status = regcomp( &re, pattern, REG_EXTENDED))!= 0){
                regerror(status, &re, buf, 120);
                exit(2);
        }

        ps = string;
        printf("String to match=%s\n", ps);
        eflag = 0;

        /* extract all the matches */
        while( status = regexec( &re, ps, 1, pmatch, eflag)== 0){
                printf("match found at: %d, string=%s\n",
                        pmatch[0].rm_so, ps +pmatch[0].rm_so);
                ps += pmatch[0].rm_eo;
                printf("\nNEXTString to match=%s\n", ps);
                eflag = REG_NOTBOL;

        }
        regfree( &re);
}
```

3. The following example uses the **fnmatch** subroutine to read a directory and match file names with a pattern.

```
#include                <locale.h>
#include                <fnmatch.h>
#include                <sys/dir.h>

main(int argc, char *argv[] )
{
        char    *pattern;
        DIR     *dir;
        struct dirent    *entry;
        int     ret;


                setlocale(LC_ALL, "");

        dir = opendir(".");

        pattern = argv[1];
```

```
        if(dir != NULL){
                while( (entry = readdir(dir)) != NULL){
                        ret = fnmatch(pattern, entry->d_name,
                                FNM_PATHNAME|FNM_PERIOD);
                        if(ret == 0){
                                printf("%s\n", entry->d_name);
                        }else if(ret == FNM_NOMATCH){
                                continue ;
                        }else{
                                printf("error file=%s\n",
                                        entry->d_name);
                        }
                }
                closedir(dir);
        }
}
```

Layout (Bidirectional Text and Character Shaping) Overview

Bidirectional (BIDI) text results when texts of different direction orientation appear together. For example, English text is read from left to right. Arabic and Hebrew texts are read from right to left. If both English and Hebrew texts appear on the same line, the text is bidirectional.

Write bidirectional text according to the following guidelines:

- Arabic and Hebrew words are written from right to left. (A character string is considered a word for the purposes of sequencing in an alphanumeric environment.)
- Numbers and English quotations are written from left to right.
- Digits and their punctuation are written marks from left to right.

Bidirectional script is read from right to left and from top to bottom.

If the embedded text is contained in one line, the text is written from left to right and embedded in the bidirectional text. However, if the embedded text is split between two or more lines, the correct order must be maintained in the left to right portions to allow top to bottom reading.

For example, right-to-left text embedded in left-to-right text that is contained in one line is written as follows:

```
THERE IS txet lanoitceridib deddebme IN THIS SENTENCE.
```

Right-to-left text embedded in left-to-right text that is split between two lines is written as follows:

```
THERE IS senil owt neewteb tilps si taht txet lanoitceridib deddebme IN THIS SENTENCE.
```

Both texts maintain readability even though the embedded text is split.

Data Streams

Bidirectional text environments use the following data streams:

Visual Data Streams

The system organizes characters in the sequence in which they are presented on the screen.

If a visual data stream is presented from left to right, the first character of the data stream is on the left side of the viewport (screen, window, line, field, and so on). If the same data stream is presented on a right-to-left viewport, the initial character of the data stream is on the right.

If a language of opposite writing orientation is embedded in the visual data stream, the sequence of each text is preserved when the viewport orientation is reversed. For example, (the lowercase text represents bidirectional text) if the keystroke order is :

```
THERE IS bidirectional text IN THIS SENTENCE.
```

then the visual data stream is:

```
THERE IS txet lanoitceridib IN THIS SENTENCE.
```

This visual data stream's presentation on a left-to-right viewport is left-justified, as follows:

```
THERE IS txet lanoitceridib IN THIS SENTENCE.
-------> <----------------- ---------------->
```

The arrows indicate reading direction.

If you change the viewport orientation to right-to-left, the visual data stream is reversed, right-justified, and unreadable, as follows:

```
.ECNETNES SIHT NI bidirectional text SI EREHT
<---------------- -----------------> <-------
```

Thus, if English text is embedded in Arabic or Hebrew text, both texts are in proper reading order only on a left-to-right viewport. The same is true for Arabic or Hebrew embedded in English. Reversing the viewport orientation makes both texts unreadable.

Logical Data Streams

The system organizes characters in a readable sequence. The bidirectional presentation-management functions arrange text strings in a readable order.

If a logical data stream is presented on a left-to-right viewport, the initial character of the data stream is presented on the left side. If the same data stream is presented on a right-to-left viewport, the initial character of the data stream is presented on the right side, though it is still presented in a readable order.

If a language of opposite writing orientation is embedded in the logical data stream, the orientations of each text are preserved by the bidirectional presentation-management functions. For example, if the keystroke order is:

```
THERE IS bidirectional text IN THIS SENTENCE.
```

then the logical data stream is the same. For example:

```
THERE IS bidirectional text IN THIS SENTENCE.
```

This logical data stream's presentation on a left to right viewport (left-justified) is as follows:

```
THERE IS txet lanoitceridib IN THIS SENTENCE.
-------> <----------------- ---------------->
```

The logical data stream's presentation on a right to left viewport (right-justified) is as follows:

```
IN THIS SENTENCE. txet lanoitceridib THERE IS
----------------> <----------------- ------->
```

The logical data stream is readable on both viewport orientations.

Cursor Movement

Cursor movement on a screen containing bidirectional text is as follows:

Visual

The cursor moves from its current position left or right to the next character, or up or down to the next row. For example, if the cursor is located at the end of the first left-to-right part of a mixed sentence:

```
THERE IS_txet lanoitceridib IN THIS SENTENCE.
```

then, moving the cursor visually to the right causes it to move one character to the right, as follows:

```
THERE IS txet lanoitceridib IN THIS SENTENCE.
```

The cursor moves without regard to the contents of the text.

Logical

The cursor moves from its current position to the next or previous character in the data stream. The character may be adjacent to the cursor's position, elsewhere in the same line, or on another line on the screen. Logical cursor movement requires scanning the data stream to find the next logical character. For example, if the cursor is located at the end of the first left-to- right part of a mixed sentence:

```
THERE IS_txet lanoitceridib IN THIS SENTENCE.
```

then, moving the cursor logically to the next character causes the data stream to be scanned to find the next logical character. The cursor moves to the next logical part of the sentence, as follows:

```
THERE IS txet lanoitceridib_IN THIS SENTENCE.
```

The cursor moves according to content.

Character Shaping

Character shaping occurs when the shape of a character is dependent on its position in a line of text. In some languages, such as Arabic, characters have different shapes depending on their position in a string and on the surrounding characters.

The following characteristics determine character shaping in Arabic script:

- The written language has no equivalent to capital letters.
- The characters have different shapes, depending on their position in a string and on the surrounding characters.
- The written language is cursive. Most characters of a word are connected, as in English handwriting.
- Joined characters can form nonspacing characters. Additionally, a character can have a vowel or diacritic mark written over or under it.
- Characters can vary in length, resulting in an output of two coded shapes.

Methods of Character Shaping

Implement character shaping separately from other system components. However, character shaping should be accessible as a utility by other system components. The system may use character shaping in the following ways:

- As the user enters data into the computer, the system uses character shaping to shape the characters. The system stores these characters in their shaped format.

  This method avoids the need to use character shaping every time these characters are displayed. This method is meant for static data such as menus and help. This method requires preprocessing for proper sorting, searching, or indexing of the characters.

  The characters may need reshaping after processing for proper presentation.
- As the user enters data into the computer, the system stores the characters in their unshaped format.

  This method allows for sorting, searching or indexing of the characters. However, the system must use character shaping every time the characters are displayed.

Base shapes are isolated shapes that were not generated by character shaping. Use base shapes during editing, searching for character strings, or other text operations. Use shaping only when the text is displayed or printed. If characters are stored in their shaped form, the system must deshape them before sorting, collating, searching, or indexing. Character shapes that are not shape determined according to their position in a string are needed for specific character-handling applications as well as for communication with different coding environments.

Contextual Character Shaping

In general, contextual character shaping is the selection of the required shape of a character in a given font depending on its position in a word and its surrounding characters. The following shapes are possible:

Isolated	A character that is connected to neither a preceding nor succeeding character.
Final	A character that is connected to a preceding character but not with a succeeding character.
Initial	A character connected to a succeeding character but not with a preceding character.
Middle	A character connected to both a preceding and succeeding character.

A character may also have any of the following characteristics:

- Connecting to a preceding character.
- Connecting to a succeeding character.
- Allowing surrounding characters' connections to pass through it.

Acronyms, part numbers, and graphic characters do not need contextual character shaping. To properly enter these characters, turn off the contextual character shaping and use a specific keyboard interface for exact selection of the desired shape. Tag these characters by field, line, or control character for later presentation.

For more information about bidirectionality and character shaping, see “Layout (Bidirectional Text and Character Shaping) Overview” on page 373, and ″Introducing the Layout Library Subroutines″ (“Introducing Layout Library Subroutines”) .

Introducing Layout Library Subroutines

For information on the layout library, please see website:

www.opengroup.org

Or order ″Portable Layout Services: Context-dependent and Directional Text″

Book# C616 ISBN 1-85912-142-X January 1997

From:

The Open Group,
Publications Department,
PO Box 96,
Witney,
Oxon OX8 6PG,
England

Tel: +44 (0)1993 708731, Fax: +44 (0)1993 708732

Use of the libcur Package

Programs that use the libcur package (extension to AT&T’s libcurses package) need to make the following changes:

1. Remove the assumption that the number of bytes need to represent a character in a code set also represents the display column width of the character. Use the **wcwidth** subroutine to determine the number of display columns needed by the wide character code of a character.
2. **NLSCHAR** is redefined to be **wchar_t**.
3. The `win->_y [y][x]` has **wchar_t** encodings.
4. Programs should not assume any particular encodings on the **wchar_t**.
5. Programs should use the **addstr**, **waddstr**, **mvaddstr**, and **mvwaddstr** subroutines rather than the **addch** family of subroutines. All string arguments are in multibyte form.
6. The **addch** and **waddch** subroutines accept a **wchar_t** encoding of the character. Programs that use these subroutines should ensure that **wchar_t** are used in calling these functions. The (x,y) are incremented by the number of columns occupied by the **wchar_t** passed to these subroutines.
7. The **delch**, **wdelch**, **mvdelch**, and **mvwdelch** subroutines support delete and backspace on multibyte characters depending on the current position of (x,y). If the current (x,y) column position points to either the first or second column of a two-column character, the **delch** subroutine deletes both columns and shifts the rest of the line by the number of columns deleted.
8. The **insch**, **winsch**, **mvinsch**, and **mvwinsch** subroutines can be used to insert a **wchar_t** encoding of a character at the current (x,y) position. The line is shifted by the number of columns needed by the **wchar_t**.
9. The libcur package is modified to support box drawing characters as defined in the **terminfo** database and not assume the graphic characters in the IBM-850 code set. The libcur package supports drawing of primary and alternate box characters as defined in the **box_chars_1** and **box_chars_2** entries in the terminfo database. To use this, programs should be modified in the following fashion:

```
Drawing Primary box characters:



        wcolorout(win, Bxa);
        cbox(win);
        wcolorend(win);

        or,
        wcolorout(win, Bxa);
        drawbox(win, y,x, height, width);
        wcolorend(win);

Drawing Alternate box characters:



        wcolorout(win, Bya)
        cboxalt(win);
        wcolorend(win);

        or,

        wcolorout(win, Bya);
        drawbox(win, y, x, height, width);
        wcolorend(win);
```

`Bxa` and `Bya` refer to the primary and alternate attributes defined in the **terminfo** database.

The following macros are added in the **cur01.h** file:

```
cboxalt(win)

drawboxalt(win, y,x, height, width)
```

10. Programs that need to support input of multibyte characters should not set **_extended** to TRUE by a call to **extended(TRUE)**. When the **_extended** flag is true, the **wgetch** subroutine returns **wchar_t** encodings of the character. With multibyte characters, this encoding of **wchar_t** may conflict with predefined values for escape sequences or function keys. Avoid this conflict when using multibyte code sets by setting extended to off (**extended(FALSE)**) before input. (The default is TRUE.)

    Programs that do multibyte character input should do the following:

    ```
    Input routine:

    Example:

        int c, count;
        char buf[];

        extended(FALSE); /* obtain one byte at a time */
        count =0;
        while(1){
            c = wgetch();  /* get one byte at a time */
            buf[count++] = c;
            if(count <=MB_CUR_MAX)
                if(mblen(buf, count) != -1)
                    break; /* character found* /
            else
                /*Error. No character can be found */
                /* Handle this case appropriately */
                break;
        }
/* buf contains the input multibyte sequence */
/* Now handle PF keys, or any escape sequence here */
    ```

11. The **inch**, **winch**, **mvinch**, and **mvwinch** subroutines return the **wchar_t** at the current (x,y) position. Note that in the case of a double column width character, if the (x,y) point is at the first column, the **wchar_t** code of the double column width character is returned. If the (x,y) point is at the second column, WEOF is returned.

Code Set Overview

To understand code sets, it is necessary to first understand *character sets*. A character set is a collection of predefined characters based on the specific needs of one or more languages without regard to the encoding values used to represent the characters. The choice of which code set to use depends on the user's data processing requirements. A particular character set can be encoded using different encoding schemes. For example, the ASCII character set defines the set of characters found in the English language. The Japanese Industrial Standard (JIS) character set defines the set of characters used in the Japanese language. Both the English and Japanese character sets can be encoded using different code sets.

The ISO2022 standard defines a coded character set as a set of precise rules that defines a character set and the one-to-one relationship between each character and it's bit pattern. A code set defines the bit patterns that the system uses to identify characters.

A *code page* is similar to a code set with the limitation that a code-page specification is based on a 16-column by 16-row matrix. The intersection of each column and row defines a coded character.

The following code sets are supported:

- Support for industry-standard code sets is provided. The **ISO8859** family of code sets provides a range of single-byte code set support that includes Latin-1, Latin-2, Cyrillic, Arabic, Greek, Hebrew, and Turkish countries. The IBM-eucJP code set is the industry-standard code set used to support the Japanese locale. The IBM-eucKR code set is the industry-standard code set used to support Korean countries. The IBM-eucTW code set is the industry-standard code set used to support Traditional Chinese countries. The IBM-eucCN code set is the industry-standard code set used to support countries using Simplified Chinese. The UTF-8 code set is a Universal Transformation Format of Unicode/ISO10646 used to support multiple languages at once (including Simplified Chinese, Traditional Chinese, and Chinese characters used in Japanese and Korean).
- **ISO8859-15** standard codeset is a replacement standard for the existing ISO8859-1 codeset that is currently in use by the western European locales, the United States, and Canada. The need for a new codeset came about as a result of the introduction of the Euro currency unit, and the need for European countries to be able to do business transactions using the Euro. In addition, ISO8859-15 contains 7 additional characters for the French and Finnish languages.
- Support is also provided for the personal computer (PC) based code sets **IBM-850**, **IBM-856**, **IBM-943**, **IBM-932**, and **IBM-1046**. IBM-850 is a single-byte code set used to support Latin-1 countries (U.S., Canada, and Western Europe). IBM-856 is a single-byte code set used to support Hebrew countries. IBM-943 and IBM-932 are multibyte code set used to support the Japanese locale. IBM-1046 is a single-byte code set used to support Arabic countries.
- **IBM-1129** is a single-byte code set used to support Vietnamese.
- **TIS-620** is a single-byte code set used to support Thai.
- **IBM-1124** is a single-byte code set used to support Ukrainian.
- Full Unicode support is provided via the **UTF-8** code set for ALL languages and territories supported by AIX. The UTF-8 code set is a Universal Transformation Format of Unicode/ISO10646 used to support multiple languages at once. The UTF-8 code set provides the most complete solution for use in environments where multiple languages and alphabets must be processed. The Unicode/UTF-8 codeset also provides full support for the common European currency (Euro).
- **IBM-1252** codeset support is provided as a compatibility option for users who require a single byte codeset environment containing the Euro currency symbol. The structure of the IBM-1252 codeset is

identical to the industry standard codeset ISO8859-1, except that additional graphic characters are added in the ISO control characters range from 0x80 through 0x9F. The Euro currency symbol is located at hexadecimal value 0x80 in the IBM-1252 codeset.

For more information on code sets, refer to these articles:

- "ASCII Characters"
- "Code Set Strategy" on page 382
- "Code Set Structure" on page 382
- "ISO Code Sets" on page 384
- "IBM PC Code Sets" on page 397

ASCII Characters

The following sections describe the 7-bit ASCII characters.

ASCII Characters in the Unique Code-Point Range

The following table lists the ASCII characters in the unique code-point range. These characters are in the range 0x00 through 0x3F.

ASCII Characters in the Unique Code-Point Range					
Symbolic Name	**Hex Value**	**Glyph**	**Symbolic Name**	**Hex Value**	**Glyph**
nul	00		space	20	blank
soh	01		exclamation-mark	21	!
stx	02		quotation-mark	22	″
etx	03		number-sign	23	#
eot	04		dollar-sign	24	$
enq	05		percent	25	%
ack	06		ampersand	26	&
alert	07		apostrophe	27	’
backspace	08		left-parenthesis	28	(
tab	09		right-parenthesis	29	)
newline	0A		asterisk	2A	*
vertical-tab	0B		plus-sign	2B	+
form-feed	0C		comma	2C	,
carriage-return	0D		hyphen	2D	-
so	0E		period	2E	.
si	0F		slash	2F	/
dle	10		zero	30	0
dc1	11		one	31	1
dc2	12		two	32	2
dc3	13		three	33	3
dc4	14		four	34	4
nak	15		five	35	5
syn	16		six	36	6
etb	17		seven	37	7

can	18		eight	38	8
em	19		nine	39	9
sub	1A		colon	3A	:
esc	1B		semicolon	3B	;
is1	1C		less-than	3C	<
is2	1D		equal-sign	3D	=
is3	1E		greater-than	3E	>
is4	1F		question-mark	3F	?

Other ASCII Characters

The following table lists the 7-bit ASCII characters that are not in the unique code-point range. These characters are in the range 0x40 through 0x7F.

Other ASCII Characters					
Symbolic Name	**Hex Value**	**Glyph**	**Symbolic Name**	**Hex Value**	**Glyph**
commercial-at	40	@	grave-accent	60	′
A	41	A	a	61	a
B	42	B	b	62	b
C	43	C	c	63	c
D	44	D	d	64	d
E	45	E	e	65	e
F	46	F	f	66	f
G	47	G	g	67	g
H	48	H	h	68	h
I	49	I	i	69	i
J	4A	J	j	6A	j
K	4B	K	k	6B	k
L	4C	L	l	6C	l
M	4D	M	m	6D	m
N	4E	N	n	6E	n
O	4F	O	o	6F	o
P	50	P	p	70	p
Q	51	Q	q	71	q
R	52	R	r	72	r
S	53	S	s	73	s
T	54	T	t	74	t
U	55	U	u	75	u
V	56	V	v	76	v
W	57	W	w	77	w
X	58	X	x	78	x
Y	59	Y	y	79	y
Z	5A	Z	z	7A	z

left-bracket	5B	[	left-brace	7B	{
backslash	5C	\	vertical-line	7C	|
right-bracket	5D	]	right-brace	7D	}
circumflex	5E	^	tilde	7E	~
underscore	5F	_	del	7F	

Code Set Strategy

Prior to AIX 3.2, IBM-850 and IBM-932 were the only supported code sets. AIX 3.2 enhanced the system code set support by adding code sets that are based on International Organization for Standardization (ISO) and industry-standard code sets. It is suggested that users use to these new code sets. The ultimate goal is to provide industry-standard code sets that satisfy the data processing needs of users.

Support for the IBM-850 codeset will be removed in future releases. Users who are currently using IBM-850 based locales should strongly consider use of the corresponding industry standard ISO8859-1 based locale. For example, users of the French IBM-850 locale (**Fr_FR**) should use the French ISO8859-1 locale (**fr_FR**).

Each locale in the system defines which code set it uses and how the characters within the code set are manipulated. Because multiple locales can be installed on the system, multiple code sets can be used by different users on the system. While the system can be configured with locales using different code sets, all system utilities assume that the system is running under a single code set.

Most commands have no knowledge of the underlying code set being used by the locale. The knowledge of code sets is hidden by the code set-independent library subroutines (NLS library), which pass information to the code set-dependent subroutines.

Because many programs rely on ASCII, all code sets include the 7-bit ASCII code set as a proper subset. Since the 7-bit ASCII code set is common to all supported code sets, its characters are sometimes referred to as the *portable character set*.

The 7-bit ASCII code set is based on the ISO646 definition and contains the control characters, punctuation characters, digits (0-9), and the English alphabet in uppercase and lowercase.

Code Set Structure

Each code set is divided into two principal areas:

Graphic Left (GL)	Columns 0-7
Graphic Right (GR)	Columns 8-F

The first two columns of each code set are reserved by International Organization for Standardization (ISO) standards for control characters. The terms C0 and C1 are used to denote the control characters for the Graphic Left and Graphic Right areas, respectively.

Note: The IBM PC code sets use the C1 control area to encode graphic characters.

The remaining 6 columns are used to encode graphic characters. Graphic characters are considered to be printable characters, while the control characters are used by devices and applications to indicate some special function.

Control Characters

Based on the ISO definition, a control character initiates, modifies, or stops a control operation. A control character is not a graphic character, but can have graphic representation in some instances. The control characters in the table below are present in all supported code sets and the encoded values of the control characters are consistent throughout the code sets.

Table 2. Code Set Control Points Table

NUL	00	Null
SOH	01	Start of header
STX	02	Start of text
ETX	03	End of text
EOT	04	End of transmission
ENQ	05	Enguiry
ACK	06	Acknowledge
BEL	07	Bell
BS	08	Backspace
HT	09	Horizontal tab
LF	0A	Line feed
VT	0B	Vertical tab
FF	0C	Form feed
CR	0D	Carrier return
SO	0E	Shift Out
SI	0F	Shift In
DLE	10	Data link escape
DC1	11	Device control 1
DC2	12	Device control 2
DC3	13	Device control 3
DC4	14	Device control 4
NAK	15	Not acknowlegde
SYN	16	Synchrous idle
ETB	17	End of trans. block
CAN	18	Cancel
EM	1	End of media
SUB	1A	Substitute character
ESC	1B	Escape character
IS4	1C	Info Separator Four
IS3	1D	Info Separator Three
IS2	1E	Info Separator Two
IS1	1F	Info Separator One

Graphic Characters

Each code set can be considered to be divided into one or more character sets, such that each character is given a unique coded value. The ISO standard reserves six columns for encoding characters and does not allow graphic characters to be encoded in the control character columns.

The internationalization of AIX is based on the assumption that all code sets can be divided into any number of character sets.

Single-Byte and Multibyte Code Sets

Code sets that use all 8 bits of a byte can support European, Middle Eastern, and other alphabetic languages. Such code sets are called single-byte code sets. This provides a limit of encoding 191 characters, not including control characters.

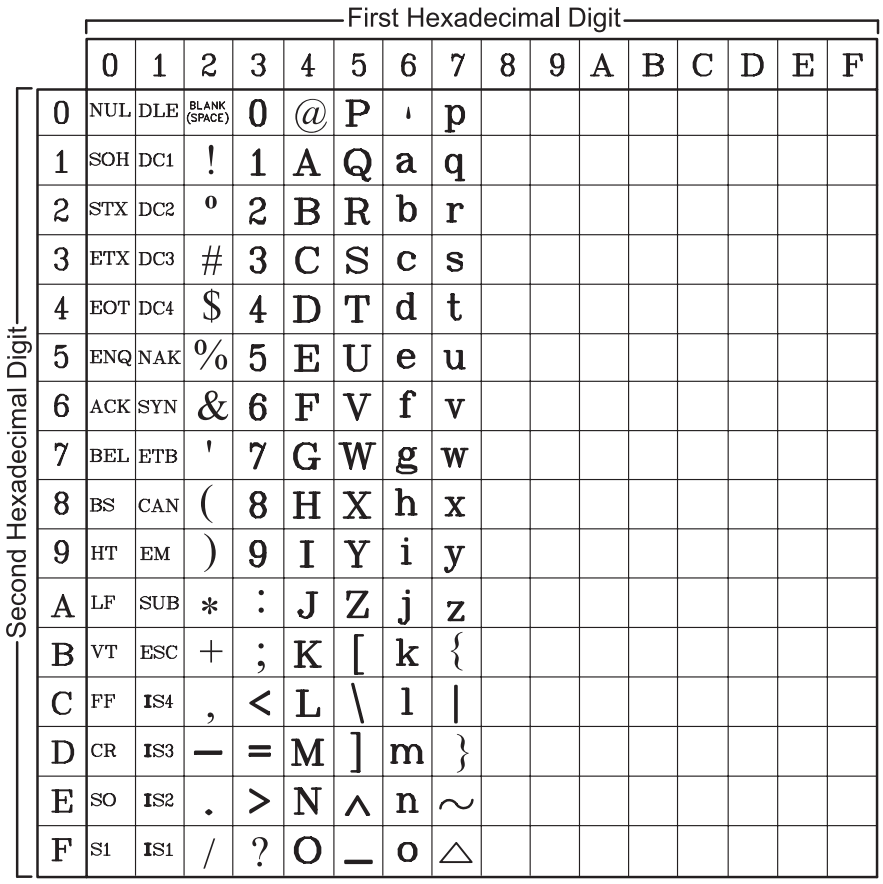
Languages that require more than 191 characters use a mixture of single-byte characters (8 bits) and multibyte characters (more than 8 bits). The system is capable of supporting any number of bits to encode a character.

ISO Code Sets

These code sets are based on definitions set by the International Organization for Standardization (ISO).

ISO646-IRV

The ″ISO646-IRV code set″ below defines the code set used for information processing based on a 7-bit encoding. The character set associated with this code set is derived from the ASCII characters.



ISO8859 Family

ISO8859 is a family of single-byte encodings based on and compatible with other ISO, American National Standards Institute (ANSI), and European Computer Manufacturer's Association (ECMA) code extension techniques. The ISO8859 encoding defines a family of code sets with each member containing its own unique character sets. The 7-bit ASCII code set is a proper subset of each of the code sets in the ISO8859 family.

While the ASCII code set defines an order for the English alphabet, the Graphic Right (GR) characters are not ordered according to any specific language. The language-specific ordering is defined by the locale.

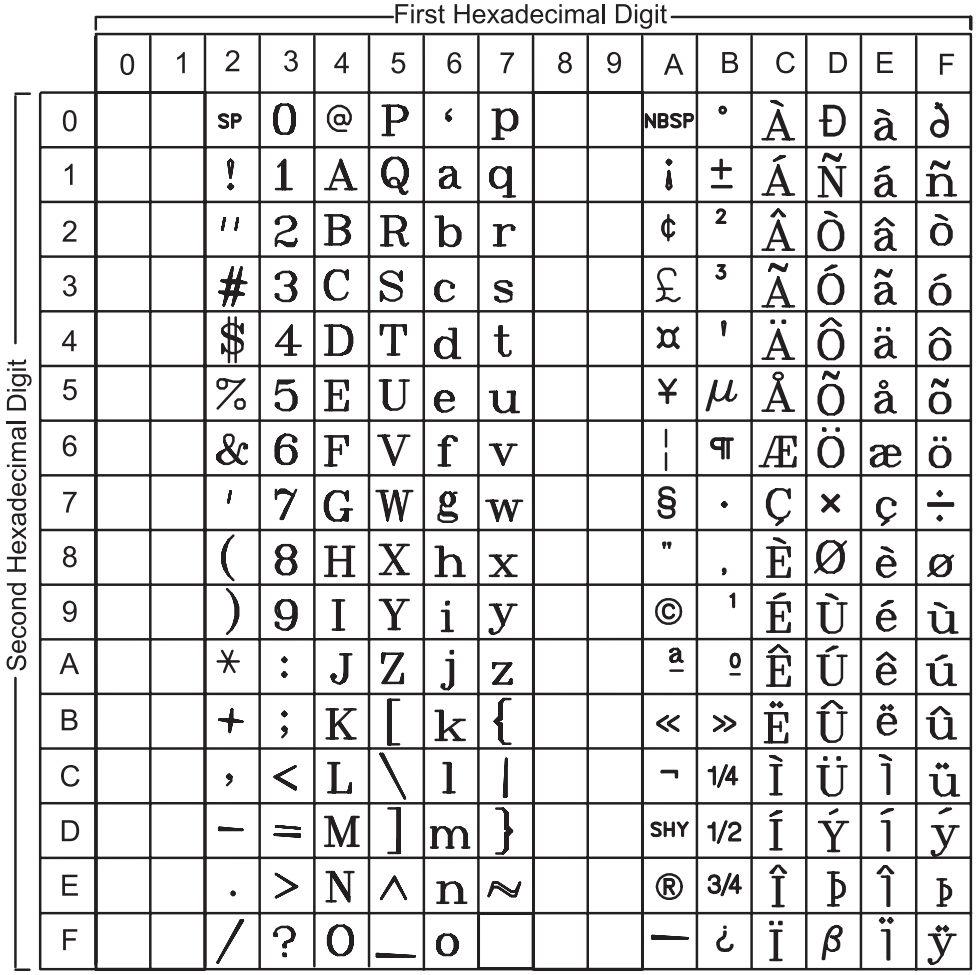
Each code set includes the ASCII character set plus its own unique character set. The ISO8859 encoding figure shows the ISO8859 general encoding scheme.

Table 3. ISO8859 Encoding

Character Encoding	Code Point	Description	Count
000xxxxx	00–1F	Controls	32
00100000	20	Space	1
0xxxxxxx	21–7E	7–bit	94
01111111	7F	Delete	1
100xxxxx	80–9F	Controls	32
10100000	A0	No-break Space	1
1xxxxxxx	A1–F	8–bit	96

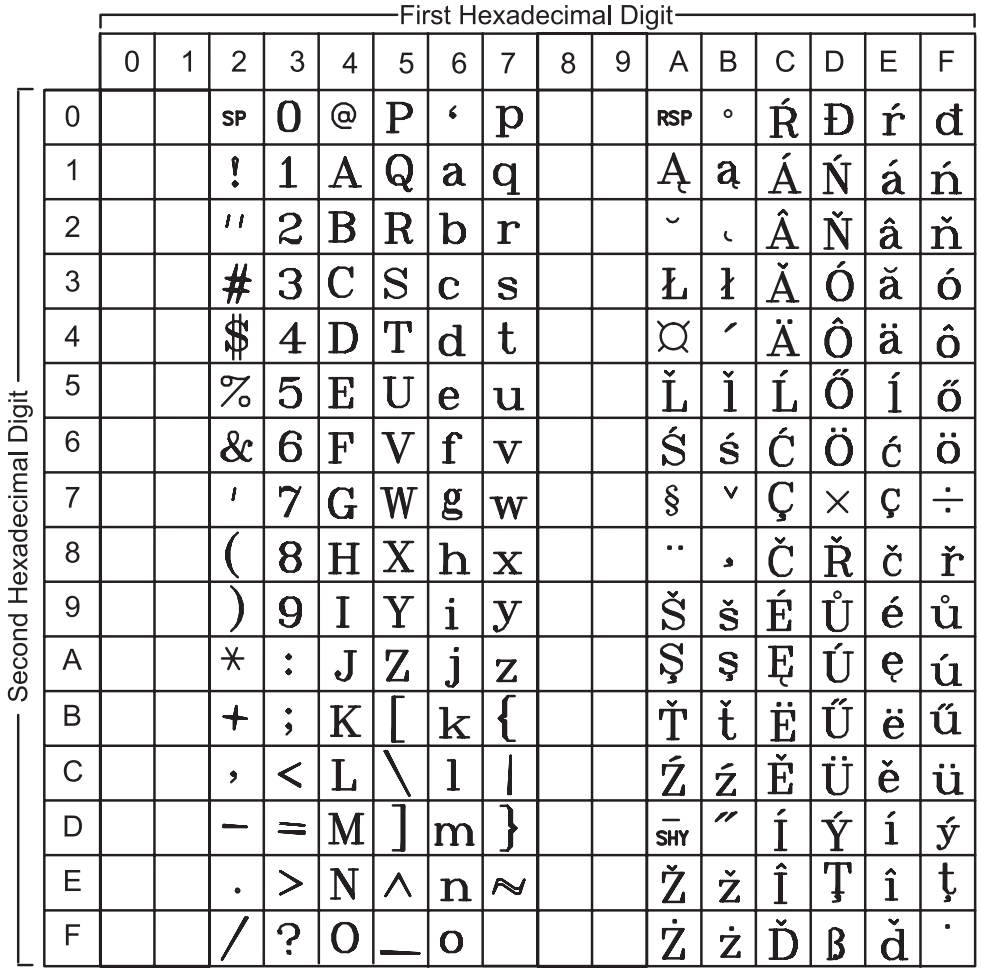
Code Set ISO8859-1

The following figure summarizes the available symbols and shows the layout of Code Set ISO8859-1. For a textual representation of this code set, see “ISO8859–1” on page 943.



Code Set ISO8859-2

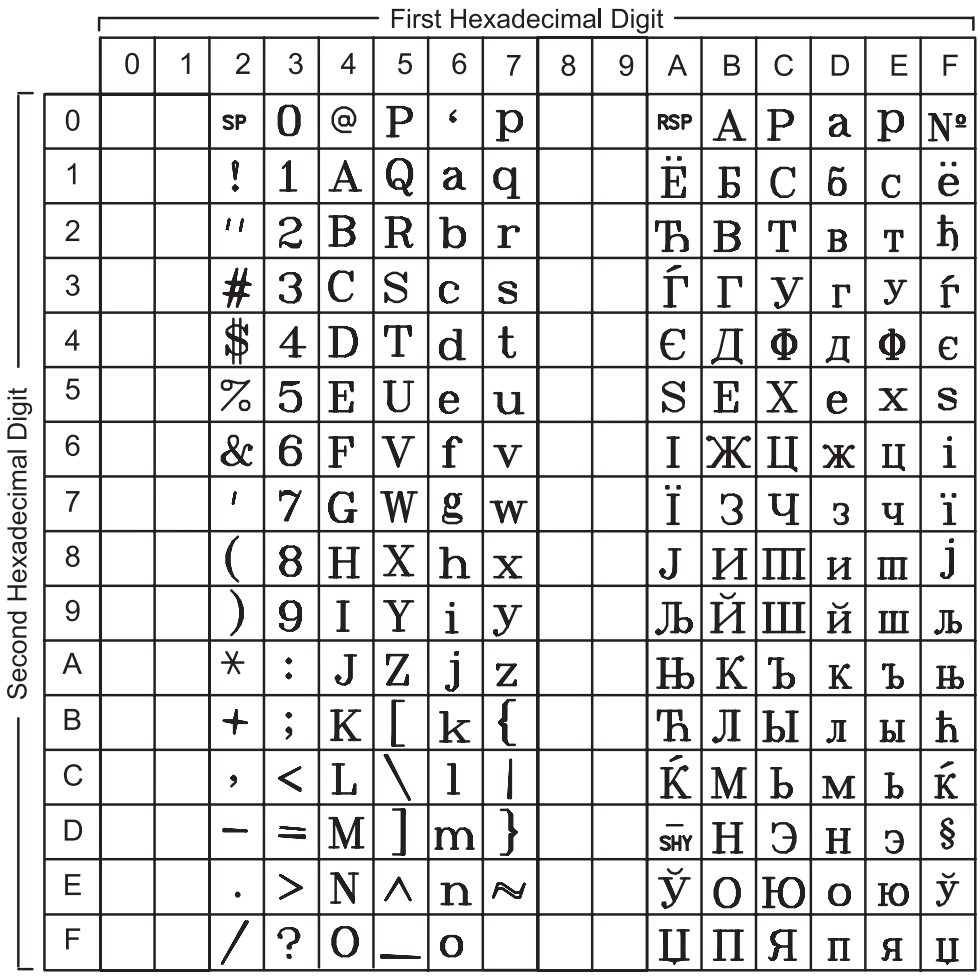
The following figure summarizes the available symbols and shows the layout of Code Set ISO8859-2. For a textual representation of this code set, see “ISO8859–2” on page 945.



Second Hexadecimal Digit \ First Hexadecimal Digit	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0			SP	0	@	P	‘	p			RSP	°	Ŕ	Đ	ŕ	đ
1			!	1	A	Q	a	q			Ą	ą	Á	Ń	á	ń
2			"	2	B	R	b	r			˘	˛	Â	Ň	â	ň
3			#	3	C	S	c	s			Ł	ł	Ă	Ó	ă	ó
4			$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô
5			%	5	E	U	e	u			Ľ	ľ	Ĺ	Ő	í	ő
6			&	6	F	V	f	v			Ś	ś	Ć	Ö	ć	ö
7			'	7	G	W	g	w			§	ˇ	Ç	×	ç	÷
8			(	8	H	X	h	x			¨	¸	Č	Ř	č	ř
9			)	9	I	Y	i	y			Š	š	É	Ů	é	ů
A			*	:	J	Z	j	z			Ş	ş	Ę	Ú	ę	ú
B			+	;	K	[	k	{			Ť	ť	Ë	Ű	ë	ű
C			,	<	L	\	l	|		Ź	ź	Ě	Ü	ě	ü
D			-	=	M	]	m	}			SHY	˝	Í	Ý	í	ý
E			.	>	N	^	n	~			Ž	ž	Î	Ţ	î	ţ
F			/	?	O	_	o				Ż	ż	Ď	ß	ď	˙

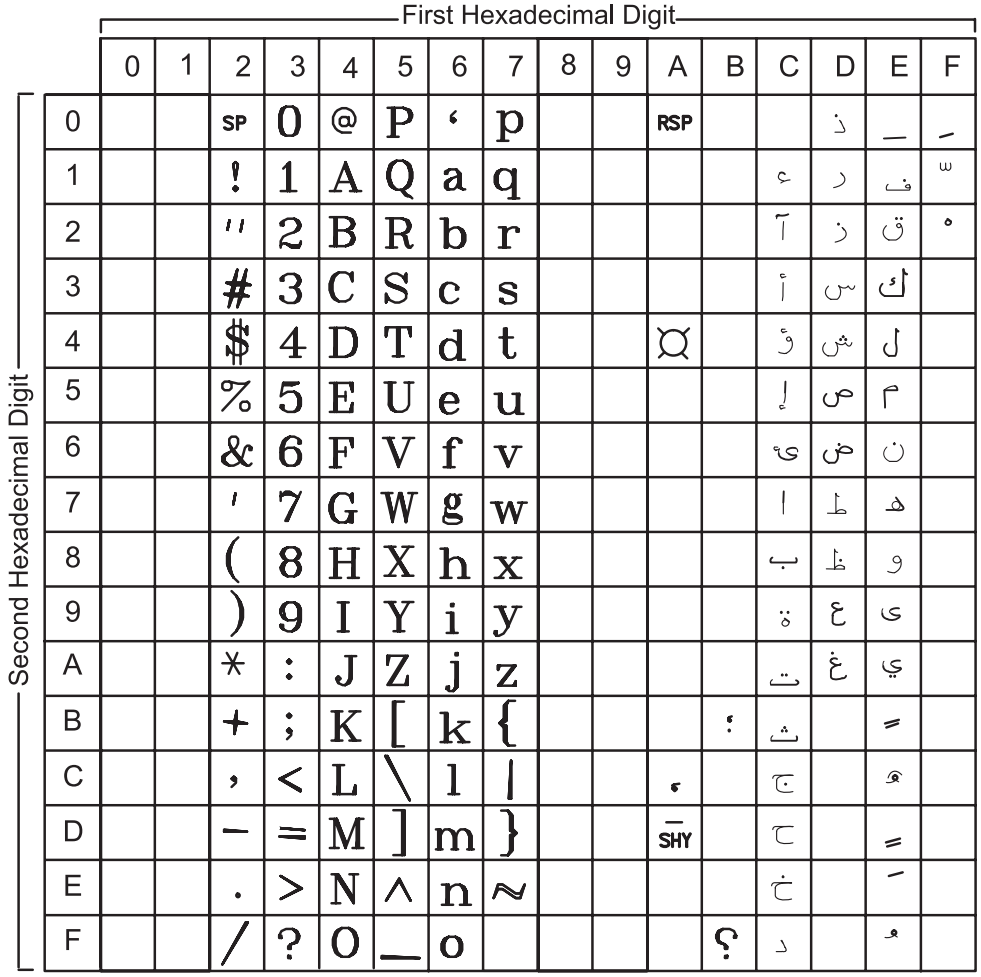
Code Set ISO8859-5

The following figure summarizes the available symbols and shows the layout of Code Set ISO8859-5. For a textual representation of this code set, see “ISO8859–5” on page 948.



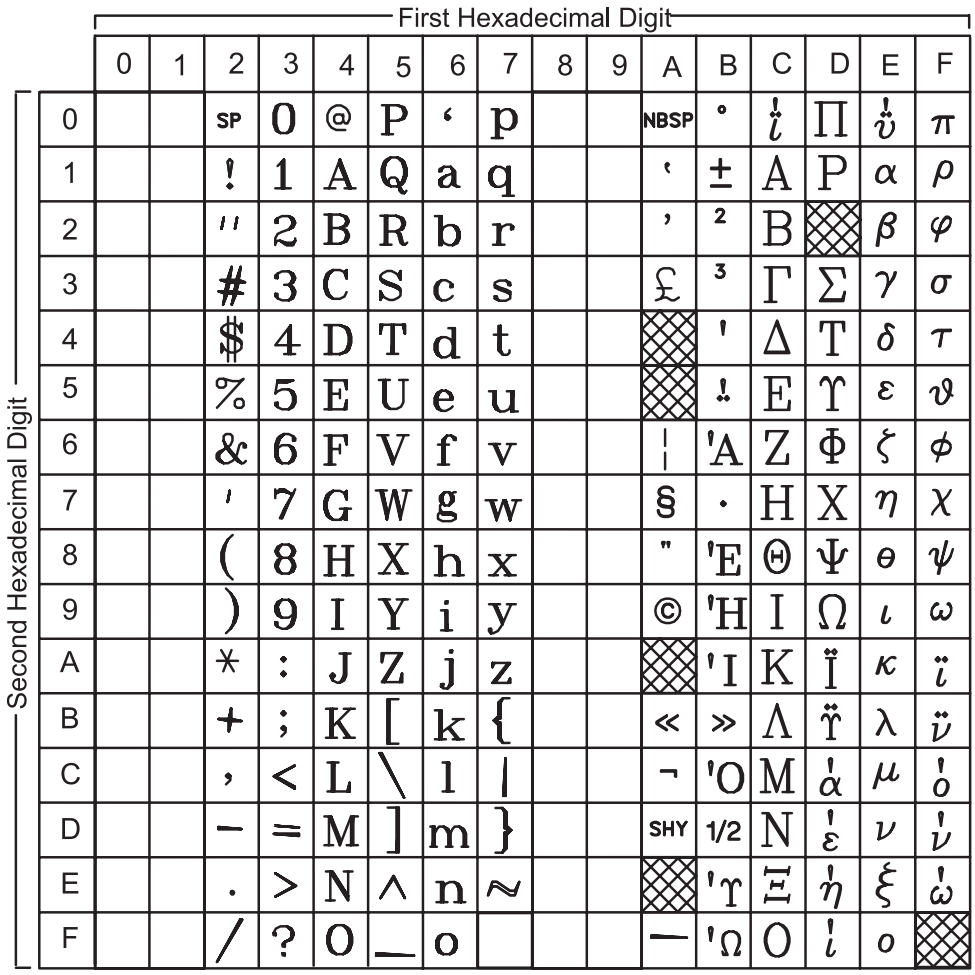
Code Set ISO8859-6

The following figure summarizes the available symbols and shows the layout of Code Set ISO8859-6. For a textual representation of this code set, see “ISO8859–6” on page 950.



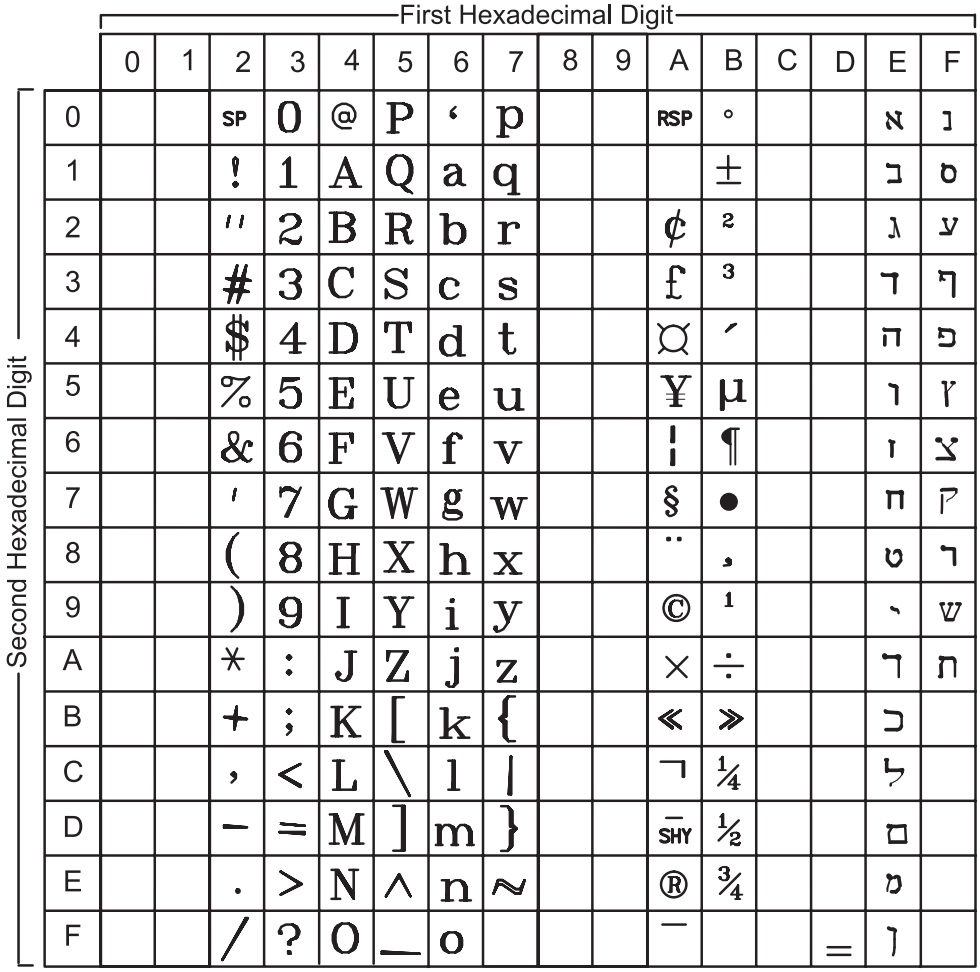
Code Set ISO8859-7

The following figure summarizes the available symbols and layout of Code Set ISO8859-7. This code set is made up of an ASCII character set plus its own unique character set. For a textual representation of this code set, see “ISO8859–7” on page 952.



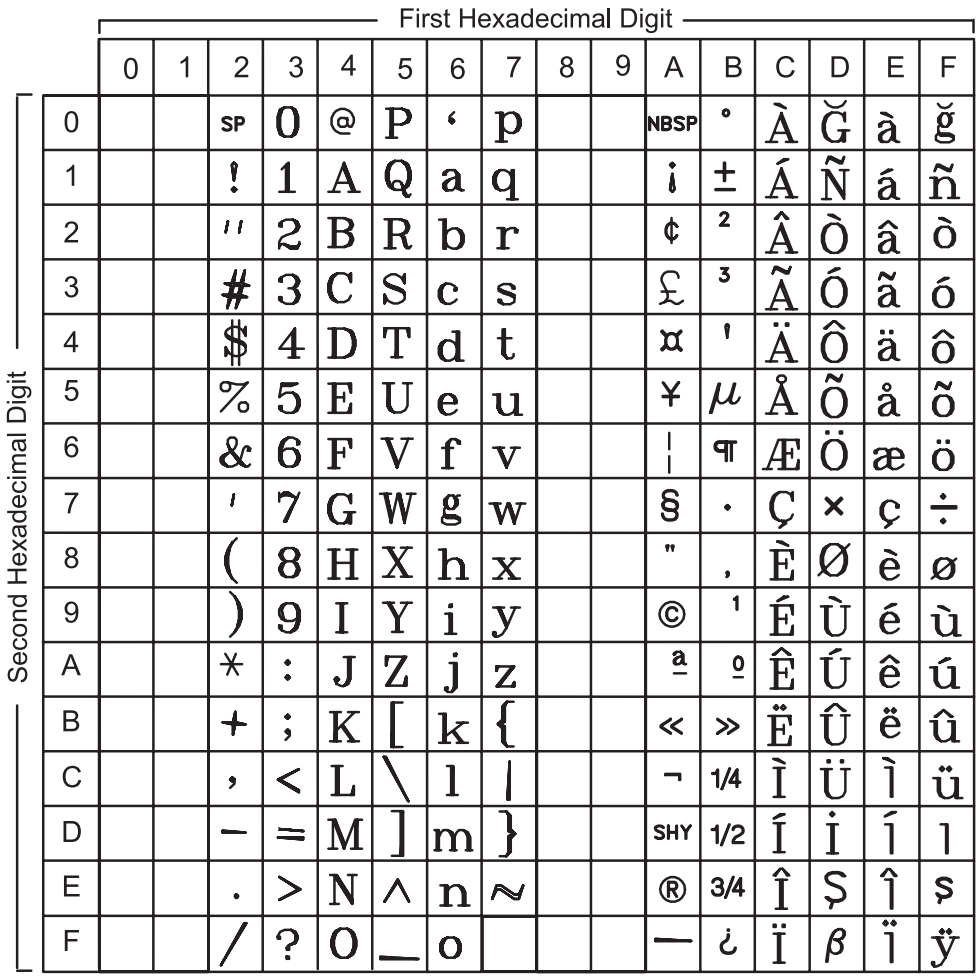
Code Set ISO8859-8

The following figure summarizes the available symbols and shows the layout of Code Set ISO8859-8. For a textual representation of this code set, see “ISO8859–8” on page 954.



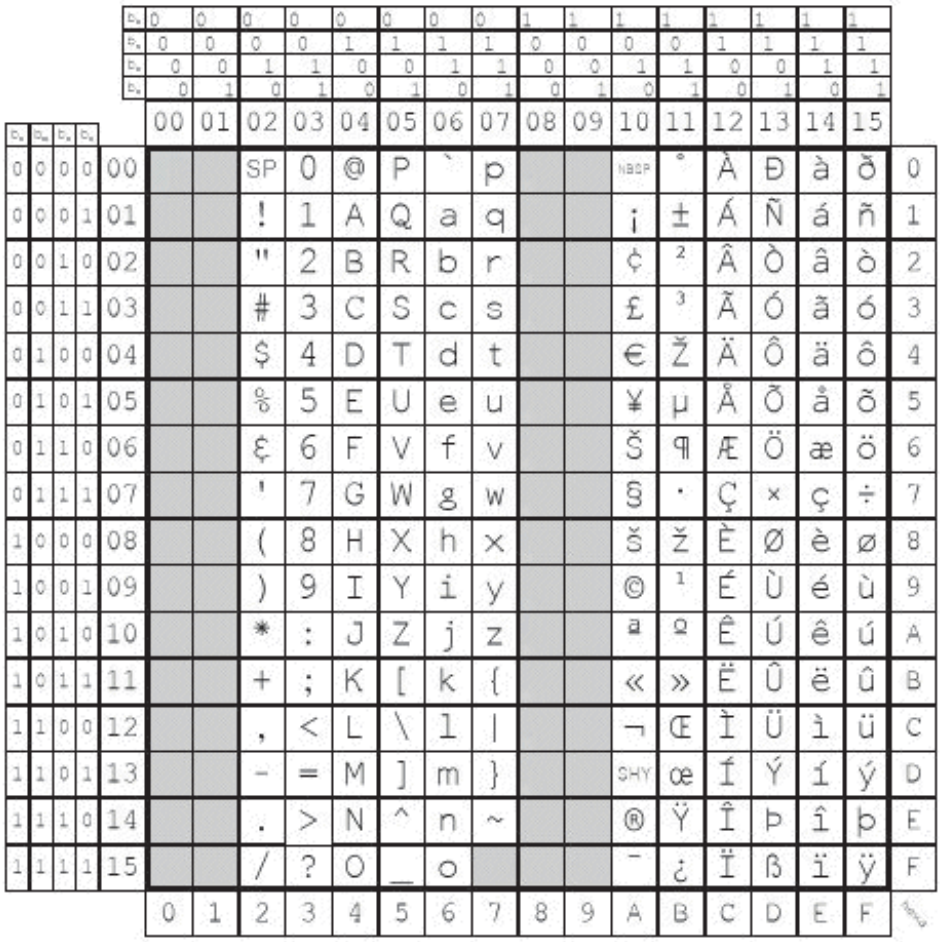
Code Set ISO8859-9

The following figure summarizes the available symbols and layout of Code Set ISO8859-9. This code set is made up of an ASCII character set plus its own unique character set. For a textual representation of this code set, see “ISO8859–9” on page 956.



Code Set ISO8859-15

The following figure summarizes the available symbols and shows the layout of Code Set ISO8859-15. For a textual representation of this code set, see “ISO8859–15” on page 958.



Extended UNIX Code (EUC) Encoding Scheme

The EUC encoding scheme defines a set of encoding rules that can support one to four character sets. The encoding rules are based on the ISO2022 definition for the encoding of 7-bit and 8-bit data. The EUC encoding scheme uses control characters to identify some of the character sets. The following table shows the basic structure of all EUC encoding.

Table 4. EUC Encoding Table

CS0	0xxxxxxx
CS1	1xxxxxxx 1xxxxxxx 1xxxxxxx 1xxxxxxx 1xxxxxxx 1xxxxxxx ...
CS2	10001110 1xxxxxxx 10001110 1xxxxxxx 1xxxxxxx 10001110 1xxxxxxx 1xxxxxxx 1xxxxxxx ...
CS3	10001111 1xxxxxxx 10001111 1xxxxxxx 1xxxxxxx 10001111 1xxxxxxx 1xxxxxxx 1xxxxxxx ...

The term EUC denotes these general encoding rules. A code set based on EUC conforms to the EUC encoding rules but also identifies the specific character sets associated with the specific instances. For example, IBM-eucJP for Japanese refers to the encoding of the Japanese Industrial Standard characters according to the EUC encoding rules.

The first set (CS0) always contains an ISO646 character set. All of the other sets must have the most significant bit (MSB) set to 1 and can use any number of bytes to encode the characters. In addition, all characters within a set must have:

- Same number of bytes to encode all characters
- Same column display width (number of columns on a fixed-width terminal)

All characters in the third set (CS2) are always preceded with the control character SS2 (single-shift 2, 0x8e). Code sets that conform to EUC do not use the SS2 control character other than to identify the third set.

All characters in the fourth set (CS3) are always preceded with the control character SS3 (single-shift 3, 0x8f). Code sets that conform to EUC do not use the SS3 control character other than to identify the fourth set.

IBM-eucJP

The EUC for Japanese is an encoding consisting of single-byte and multibyte characters. The encoding is based on ISO2022, Japanese Industrial Standard (JIS), and EUC definitions.

The IBM-eucJP code set consists of the following character sets:

JISCII	JISX0201 Graphic Left character set
JISX0201.1976	Katakana/Hiragana Graphic Right character set
JISX0208.1983	Kanji level 1 and 2 character sets
IBM-udcJP	IBM-user definable characters

The IBM-eucJP code set is also capable of supporting:

JISX0212.1990	Supplemental Kanji

The IBM-eucJP code set is encoded as follows:

- CS0 maps JISX0201 Graphic Left characters starting at the 0x00 position.
- CS1 maps the JISX0208 character set starting at the 0xa1xa1 position.

The positions 0xf5a1 through 0xfefe (940 characters) in CS1 are reserved as primary user definable character areas.

- CS2 maps the JISX0201 Graphic Right starting at the 0x8ea1 position.
- CS3 is capable of mapping JISX0212 starting at the 0x8fa1a1 position.

The positions 0x8ff5a1 through 0x8ffefe in CS3 (940 characters) are reserved as secondary user definable character areas.

- The positions 0x8feea1 through 0x8ff4fe in CS3 (658 characters) are reserved for future system use. Therefore, users should not use this area.

IBM-eucCN

The EUC for the Simplified Chinese language is an encoding consisting of characters that contain 1 or 2 bytes. The EUC encoding is based on ISO2022, GB2312 as defined by the People's Republic of China, and multibyte character definitions unique to the manufacturer.

The current GB2312 defines 6,763 Simplified Chinese characters and 682 symbols. The IBM-eucCN is based upon a concept of one plane containing up to 94x94 characters. The encoding values of these characters range from 0xa1a1 to 0xfefe.

The GB2312 is mapped into the CS1 of EUC. Specifically, the IBM-eucCN consists of the following character sets:

ISO0646-IRV 7-bit ASCII character set, Graphic Left.

GB2312.1980 Contains 7445 characters. It occupies positions 0xa1a1 to 0xfedf (some user-defined characters scattered in 0xa1a1 to 0xfedf).

IBM-udcCN Scattered in GB. It occupies positions Oxa1a1 to Oxfedf. The actual values are:

```
a2a1 -- a2b0    a1e3 -- a2e4  a1ef -- a2f0
a2fd -- a1fe    a4f4 -- a4fe  a5f7 -- a5fe
a6b9 -- a6c0    a6d9 -- a6fe  a7c2 -- a7d0
a7f2 -- a7fe    a8bb -- a8c4  a8ea -- a9a3
a9f0 -- affe    a7fa -- d7fe  f8a1 -- fedf
```

IBM-sbdCN Scattered in GB. It occupies positions 0xfee0 to 0xfefe.

GBK

GBK stands for Guo (national) Biao (Standard) Kuo (Extension). GBK expands the national ″Industry GB″ definition to contain all 20, 902 Han Characters defined in Unicode and additional DBCS symbols defined in Big-5 code (Traditional Chinese PC defacto standard). Restated, GBK defined all DBCS characters and symbols in use on both sides of the Taiwan Strait. Currently, GBK is a Normative Annex of GB13000 (PRC Unicode Standard) and is being positioned as an interim step for migration to Unicode.

Locale	Code Set	Description
Zh_CN	GBK	Simplified Chinese, GBK Locale

Code Range	Words	Marks
A1A1-A9FE	846	GB2312, GB12345 (GBK/1)
A840-A9A0	192	Big5, Symbols (GBK/5)
B0A1-F7FE	6768	GB2312 (GBK/2)
8140-A0FE	6080	GB13000 (GBK/3)
AA40-FEA0	8160	GB13000 (GBK/4)
AAA1-AFFE	564	User defined 1
F8A1-FEFE	658	User defined 2
A140-A7A0	672	User defined 3

IBM-eucTW

The EUC for the Traditional Chinese language is an encoding consisting of characters that contain 1, 2 and 4 bytes. The EUC encoding is based on ISO2022, the Chinese National Standard (CNS) as defined by the Republic of China and multibyte character definitions unique to the manufacturer.

The current CNS defines 13,501 Chinese characters and 684 symbols. The IBM-eucTW is based upon a concept of 15 planes, each containing up to 8836 (94x94) characters. The encoding values of these characters range from 0xa1a1 to 0xfefe. Characters have presently been defined for only 4 of the planes, with the other planes being reserved for future expansion.

The 15 planes are mapped into the CS1 and CS2 of EUC, with the CS2 of EUC consisting of 14 planes. Specifically, the IBM-eucTW consists of the following character sets:

ISO646-IRV	7-bit ASCII character set, Graphic Left.
CNS11643.1986-1	Plane 1, containing 6085 characters (5401+684). This plane uses positions 0ax1a1-0xc2c1 and 0xc4a1-0xfdcb.
CNS11643.1986-2	Plane 2, containing 7650 characters. This plane occupies positions 0x8ea2a1a1-0x8ea2f2c4.
CNS11643.1992-3	Plane 4, containing 7298 characters. This plane occupies positions 0x8ea4a1a1-0x8ea4eedc.
IBM-udcTW	Plane 12, containing 6204 characters. This plane is reserved for the User Defined Characters (udc) areas. It occupies the positions 0x8eaca1a1-0x8ea2f2c4.
IBM-sbdTW	Plane 13, containing 325 characters. This plane is reserved for symbols unique to the manufacturer. It occupies positions 0xeada1a1-0x8eada4cb.

Planes 3-11 are expected to occupy positions 0x8ea3xxxx to 0x8eabxxxx. Planes 14-15 are expected to occupy positions 0x8eaexxxx to 0x8eafxxxx.

Big5

The Traditional Chinese big5 locale, **Zh_TW**, code set is the most commonly used code set in the PC field which is used to support countries using Traditional Chinese.

Big5 code set defines 13056 characters and 1004 symbols. It includes 684 symbols in CNS11643.192, as well as 325 IBM unique symbols.

Locale	Code Set	Description
Zh_TW	Big5 (IBM-950)	Traditional Chinese, Big5 Locale

Code Range for Big5 Locale::

Plan	Code Range	Description
1	A140H - A3E0H	Symbol and Chinese Control Code
1	A440H - C67EH	Commonly Used Characters
2	C940H - F9D5H	Less Commonly Used Characters
UDF	FA40H - FEFE	User-Defined Characters
	8E40H - A0FEH	User-Defined Characters
	8140H - 8DFEH	User-Defined Characters
	8181H - 8C82H	User-Defined Characters
	F9D6H - F9F1H	User-Defined Characters

Code Set	Words	Code Range	Marks
Commonly Used Area	5841	A140-C67E	
Less Commonly Used Area	7652	C940-F9D5	
ET Unique Area (1)	308	C6A1-C878	
ET Unique Area (2)	7	C8CD-C8D3	
IBM Unique Area	251	F286-F9A0	Low-Byte Range 81-A0
User-Defined Area (1)	785	FA40-FEFE	
User-Defined Area (2)	2983	8E40-A0FE	
User-Defined Area (3)	2041	8140-8DFE	
User-Defined Area (4)	354	8181-8C82	Low-Byte Range 81-AQ

Code Set	Words	Code Range	Marks
User-Defined Area (5)	41	F9D6-F9FE	

IBM-eucKR

The EUC for the Korean language is an encoding consisting of single-byte and multibyte characters. The encoding is based on ISO2022, Korean Standard Code set and EUC definitions.

The Korean EUC code set consists of two main character groups:

- ASCII (English)
- Hangul (Korean characters)

The Hangul code set includes Hangul and Hanja (Chinese) characters. One Hangul character can be comprised of several consonants and vowels. However, most Hangul words can be expressed in Hanja. Each Hanja character has its own meaning and is more specific than Hangul.

The IBM-eucKR consists of the following character sets:

ISO646-IRV 7-bit ASCII character set, Graphic Left
KSC5601.1987-0 Korean Graphic Character Set, Graphic Right

IBM PC Code Sets

IBM PC code sets are the code sets originally supported on the IBM PC systems and AIX. The IBM PC code sets assign graphic characters to the Control One (C1) control area. Applications that depend on these control characters can not support these code sets.

The ASCII characters are encoded with the most significant bit (MSB) zero in positions 0x20-0x7e. The extended Latin 1 combined with the IBM PC unique characters sets make up the extended set of characters which are encoded in positions 0x80-0xfe. The table below shows the location of the control, ASCII, and extended characters for the IBM-850 code set.

Character Encoding	Code Points	Description	Count
000xxxxx	00–1F	Controls	32
00100000	20	Space	1
0xxxxxxx	21–7E	7–bit	94
01111111	7F	Delete	1
1xxxxxxx	80–FE	8–bit	17
11111111	FF	All ones	1

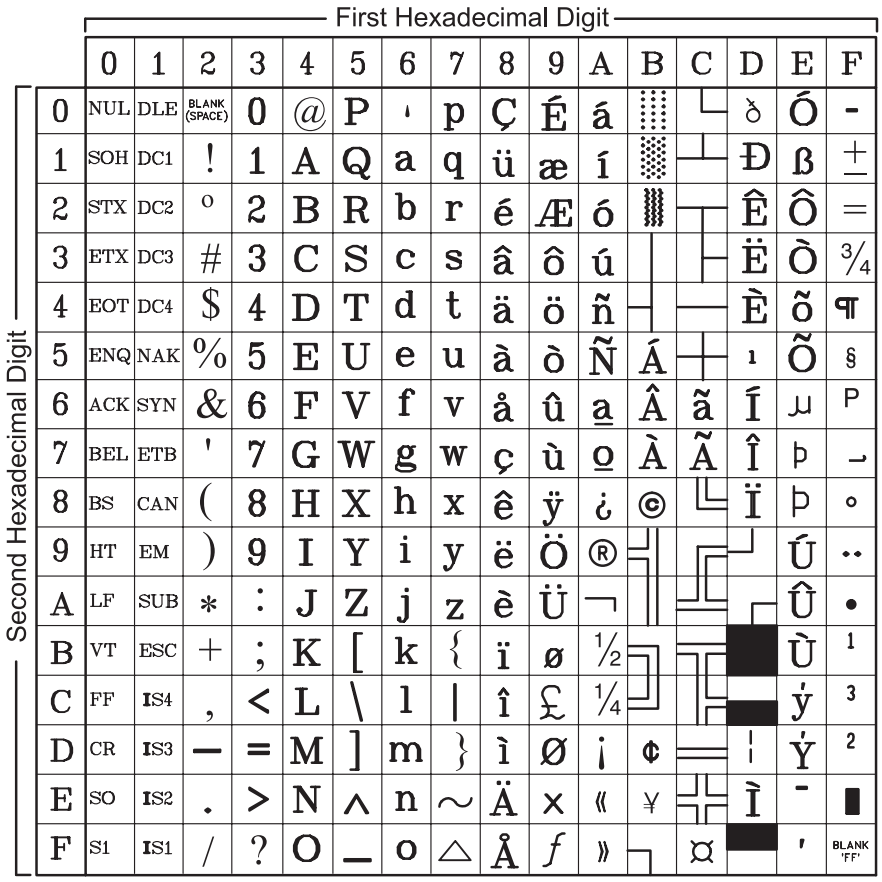
The IBM PC unique character set includes the following:

IBM PC Unique Character Set	
Symbol	**Return Code**
Florin sign	0x9f
Quarter-hashed	0xb0
Half-hashed	0xb1
Full-hashed	0xb2

Vertical bar	0xb3
Right-side middle	0xb4
Double right-side middle	0xb9
Double vertical bar	0xba
Double upper right-corner box	0xbb
Double lower right-corner box	0xbc
Upper right-corner box	0xbf
Lower left-corner box	0xc0
Bottom-side middle	0xc1
Top-side middle	0xc2
Left-side middle	0xc3
Center-box bar	0xc4
Intersection	0xc5
Double lower left-corner box	0xc8
Double upper left-corner box	0xc9
Double bottom-side middle	0xca
Double top-side middle	0xcb
Double left-side middle	0xcc
Double center-box bar	0xcd
Double intersection	0xce
Small i dotless	0xd5
Lower right-corner box	0xd9
Upper left-corner box	0xda
Bright character cell	0xdb
Bright character cell - lower half	0xde
Bright character cell - upper half	0xdf
Overbar	0xee
Middle dot, Product dot	0xfa
Vertical solid rectangle	0xfe

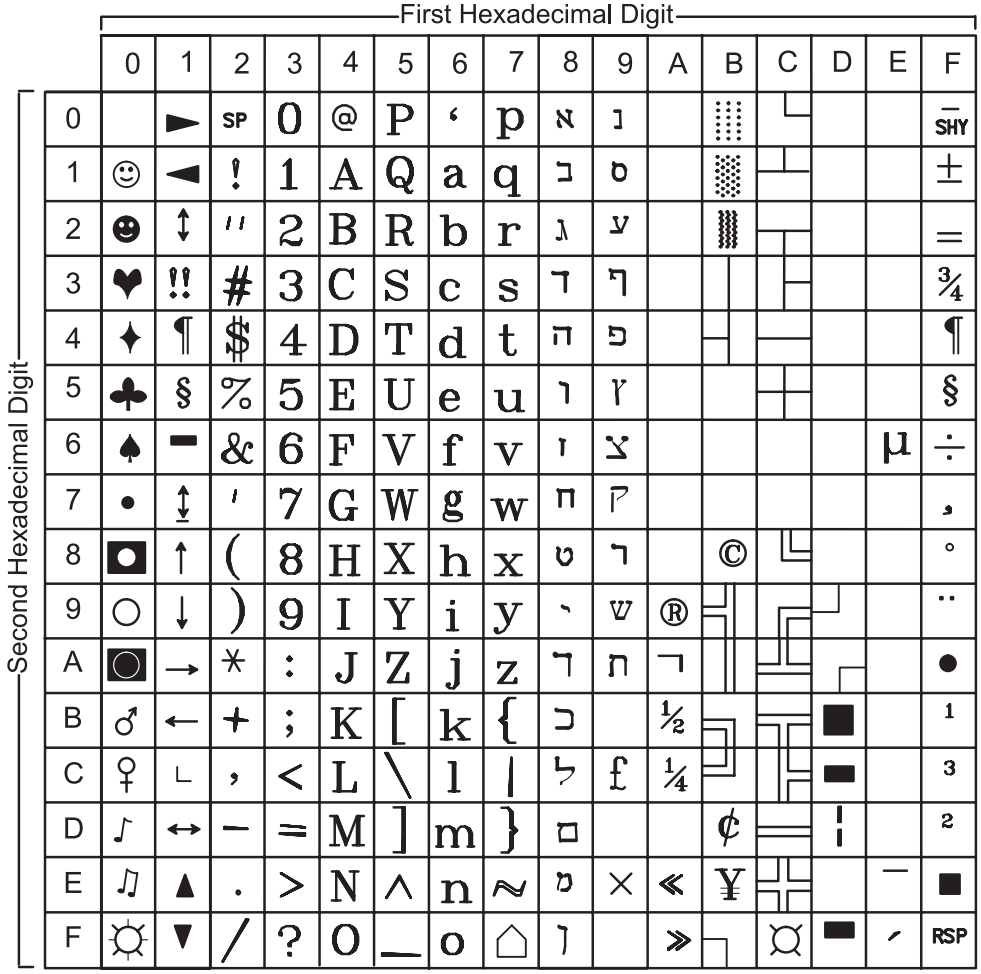
IBM-850

The following figure summarizes the available symbols and shows the layout of Code Set IBM-850. For a textual representation of this code set, see “IBM-850” on page 961.



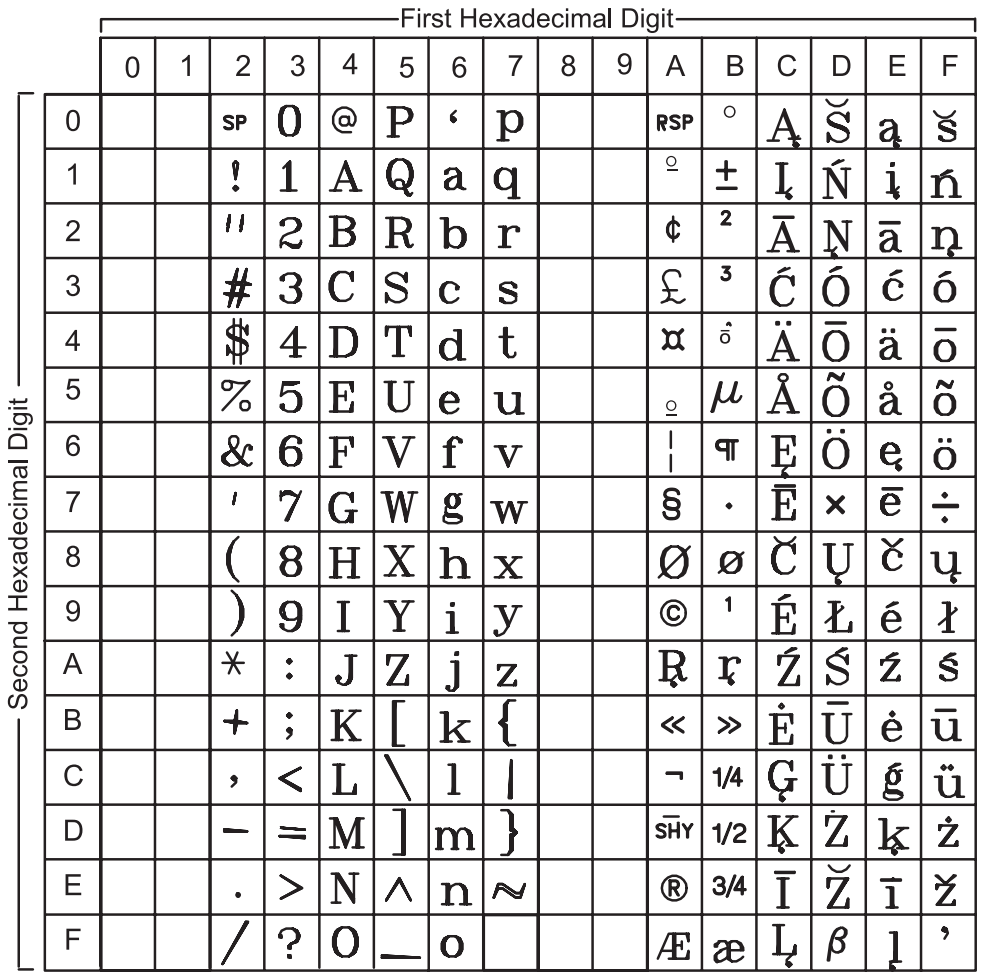
IBM-856

The following figure summarizes the available symbols and shows the layout of Code Set IBM-856. For a textual representation of this code set, see “IBM-856” on page 964.



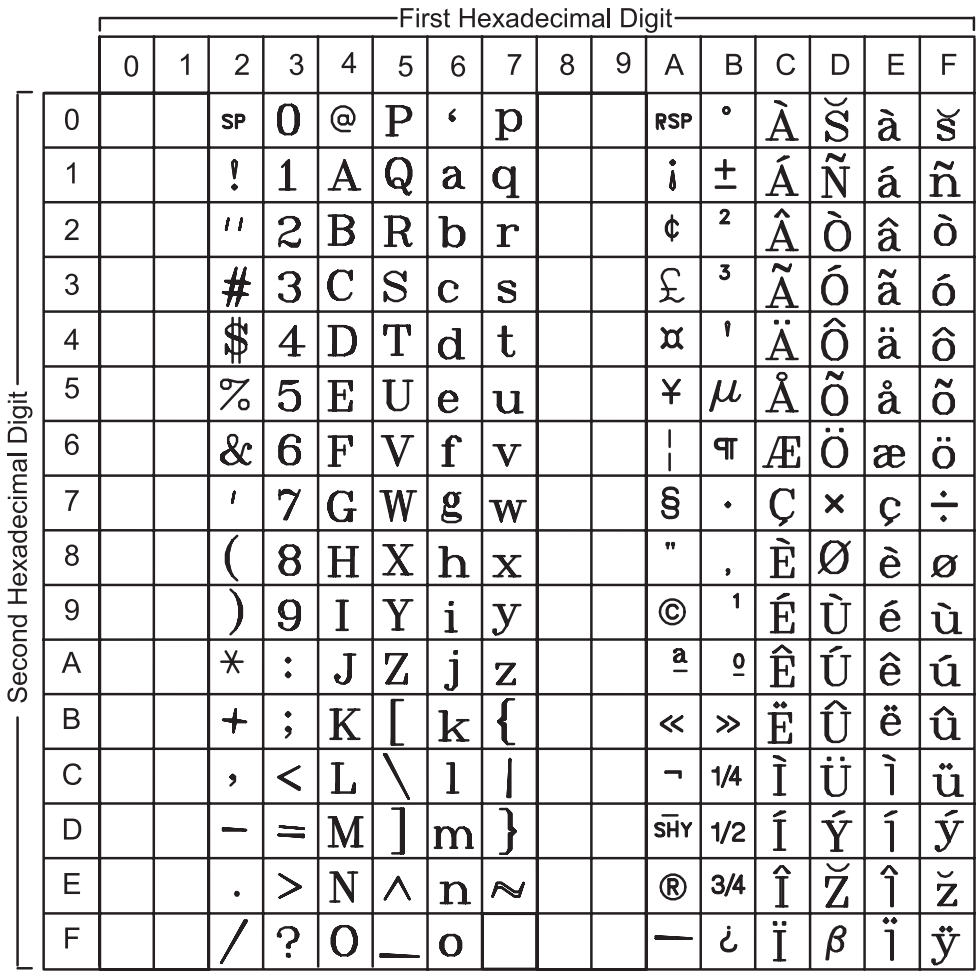
IBM-921

The following figure summarizes the available symbols and shows the layout of Code Set IBM-921. For a textual representation of this code set, see “IBM-921” on page 966.



IBM-922

The following figure summarizes the available symbols and shows the layout of Code Set IBM-922. For a textual representation of this code set, see “IBM-922” on page 969.



IBM-943 and IBM-932

Each of the Japanese IBM PC code sets are an encoding consisting of single-byte and multibyte coded characters. The encoding is based on the IBM PC code set and places the JIS characters in shifted positions. This is referred to as *Shift-JIS* or SJIS.

IBM-943 is newer code set for the Japanese locale than IBM-932. IBM-943 is a compatible code set for the Japanese Microsoft Windows environment. This code set is known as **1983 ordered shift-JIS**. The difference between IBM-932 and IBM-943 is as follows

- Old JIS sequence (1978 ordered) is applied for IBM-932 while new JIS sequence (1983 ordered) is applied for IBM-943.
- NEC selected characters are added to IBM-943.
- NEC's IBM selected characters are added to IBM-943.

The IBM-932 consists of the following character sets:

JISCII	JISX0201 Graphic Left character set
JISX0201.1976	Katakana/Hiragana Graphic Right character set
JISX0208.1983	Kanji level 1 and 2 character sets
IBM-udcJP	IBM user-definable characters

The IBM-943 consists of the following character sets:

JISCII	JISX0201 Graphic Left character set
JISX0201.1976	Katakana/Hiragana Graphic Right character set
JISX0208.1990	Kanji level 1 and 2 character sets
IBM-udcJP	IBM user-definable characters and NEC's IBM selected characters and NEC selected characters

The first byte of each character is used to determine the number of bytes for a given character. The values 0x20-0x7e and 0xa1-oxdf are used to encode JISX0201 characters, with exceptions. The positions 0x81-0x9f and 0xe0-0xfc are reserved for use as the first byte of a multibyte character. The JISX0208 characters are mapped to the multibyte values starting at 0x8140. The second byte of a multibyte character can have any value. The Shift-JIS table shows where these characters are located on the code set.

Table 5. Shift-JIS (IBM-943 and IBM-932) Encoding Scheme for Japanese

Character Encoding	Code Point	Description	Count
000xxxxx	00–1f	Controls	32
00100000	20	Space	1
0xxxxxxx	21–7E	7–bit ASCII	94
01111111	7F	Delete	1
10000000	80	Undefined	1
100xxxxx 01xxxxxx	[81–9F] [40–7E]	Double byte	1953
100xxxxx 1xxxxxxx	[81–9F] [80–FC]	Double byte	3975
10100000	A0	Undefined	1
1xxxxxxx	A1–DF	8–bit single byte	63
111xxxxx 01xxxxxx	[E0–FC] [40–7E]	Double byte	1827
111xxxxx 1xxxxxxx	[E0–FC] [80–FC]	Double byte	3625
11111101	FD	Undefined	1
11111110	FE	Undefined	1
11111111	FF	Undefined	1

The following table shows the DBCS part of IBM-943.

Table 6. DBCS Portion of IBM-943

Code Point	Description
[81–84] [40–7E] and [81–84] [80–F0]	JIS X 0208 (Non-Kanji)
[87] [40–7E] and [87] [80–F0]	NEC selected characters
[89–98] [40–7E] and [88] [9F-F0], [89–97] [80–F0], [98] [80–9F]	JIS X0208 (Level-1 Kanji)
[99–9F] [40–7E] and [98] [9F-F0], [99–9F] [80–F0]	JIS X0208 (Level-2 Kanji)
[E0–EA] [40–7E] and [E0–EA] [80–F0]	JIS X0208 (Level-2 Kanji)
[ED–EE] [40–7E] and [ED–EE] [80–F0]	NEC IBM selected characters
[F0–F9] [40–7E] and [F0–F9] [80–F0]	User defined characters
[FA] [40–5C]	IBM selected characters (non-Kanji)

Table 6. DBCS Portion of IBM-943 (continued)

[FA] [5C-7E], [FB-FC] [40–7E] and [FA-FC] [80–F0]	IBM selected characters (Kanji)

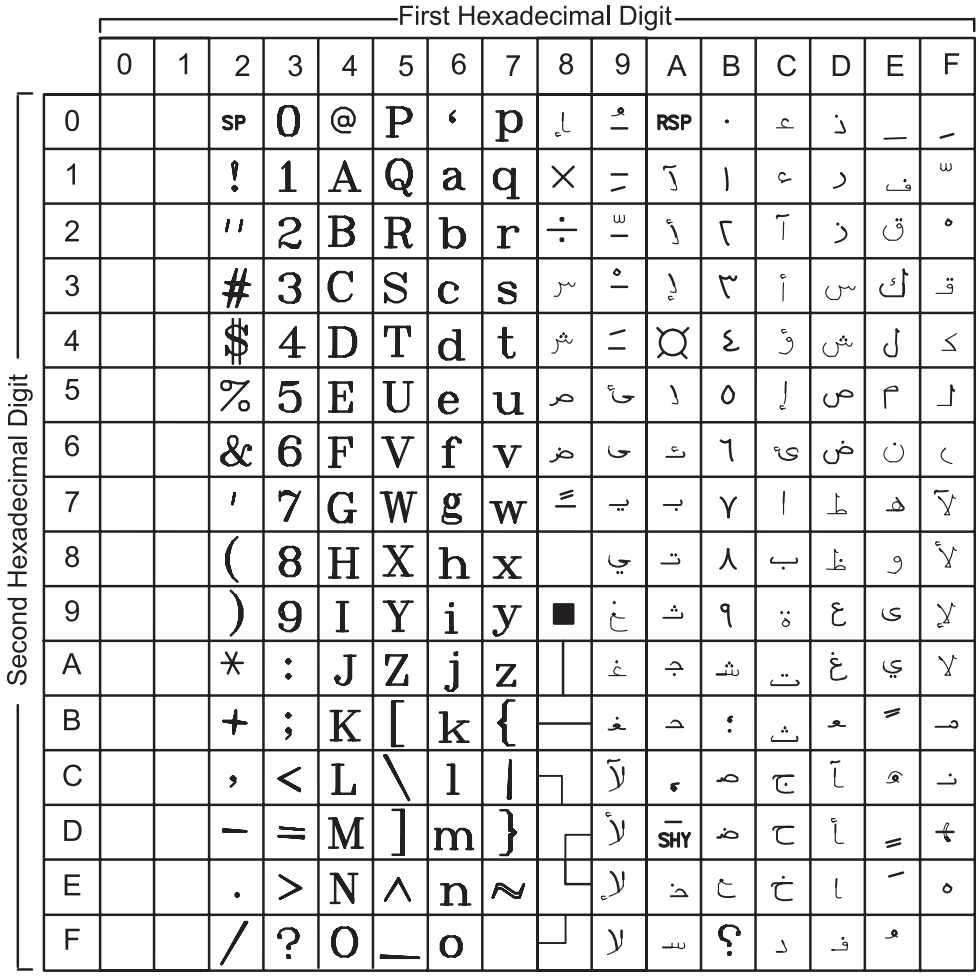
The following table shows the DBCS part of IBM-932.

Table 7.

Code Point	Description
[81–98] [40–7E] and [81–97] [80–FC], [98] [80–9F]	JIS X 0208 (Level-1 Kanji)
[99–9F] [40–7E] and [98] [9F-FC], [99–9F] [80–FC]	JIS X 0208 (Level-2 Kanji)
[E0–EF] [40–7E] and [E0–EF] [80–FC]	JIS X 0208 (Level-2 Kanji)
[F0–F9] [40–7E] and [F0–F9] [80–FC]	User defined characters
[FA–FC] [40–7E] and [FA–FC] [80–FC]	IBM selected characters

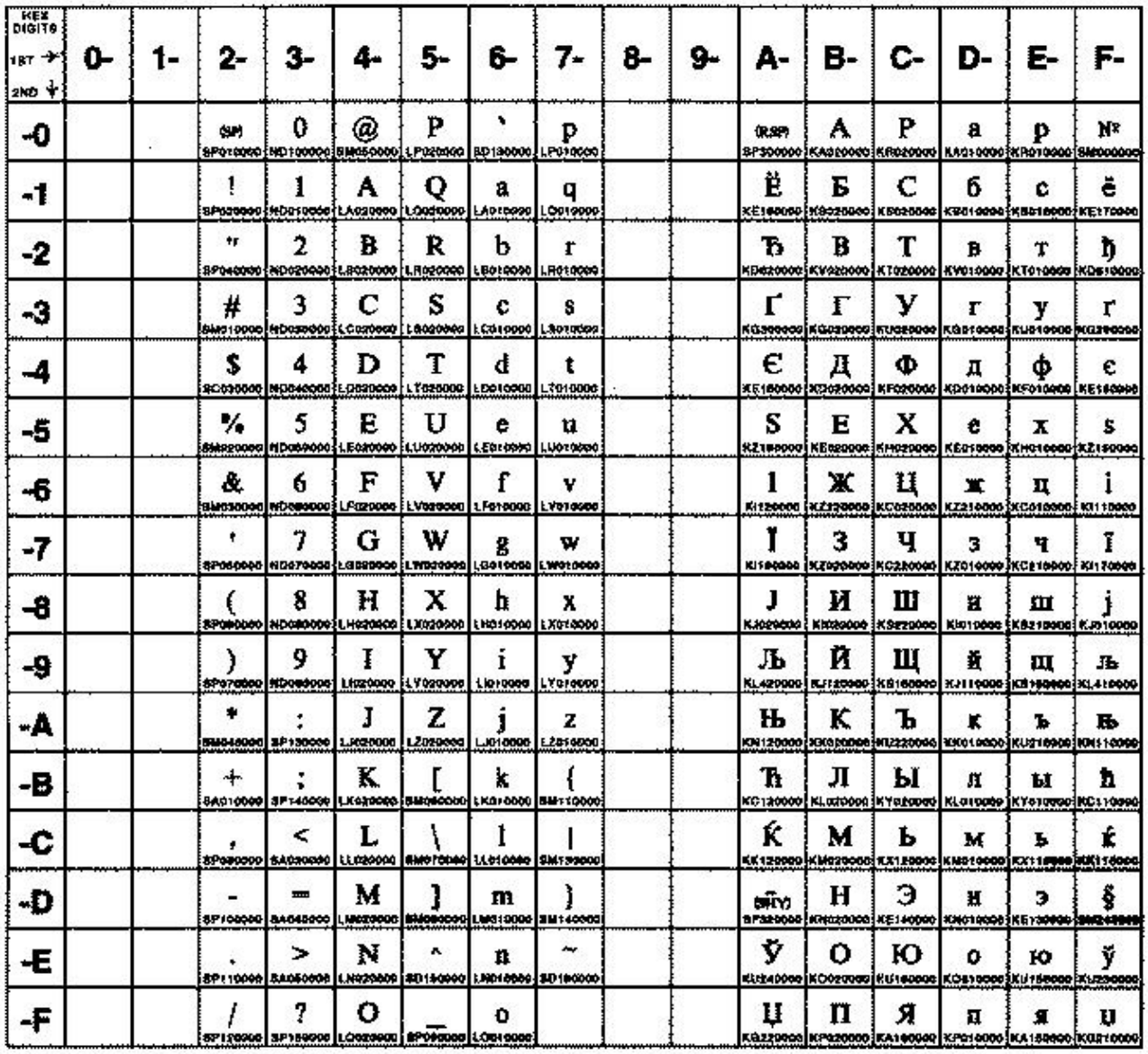
IBM-1046

The following figure summarizes the available symbols and shows the layout of Code Set IBM-1046. For a textual representation of this code set, see “IBM-1046” on page 971.



IBM-1124

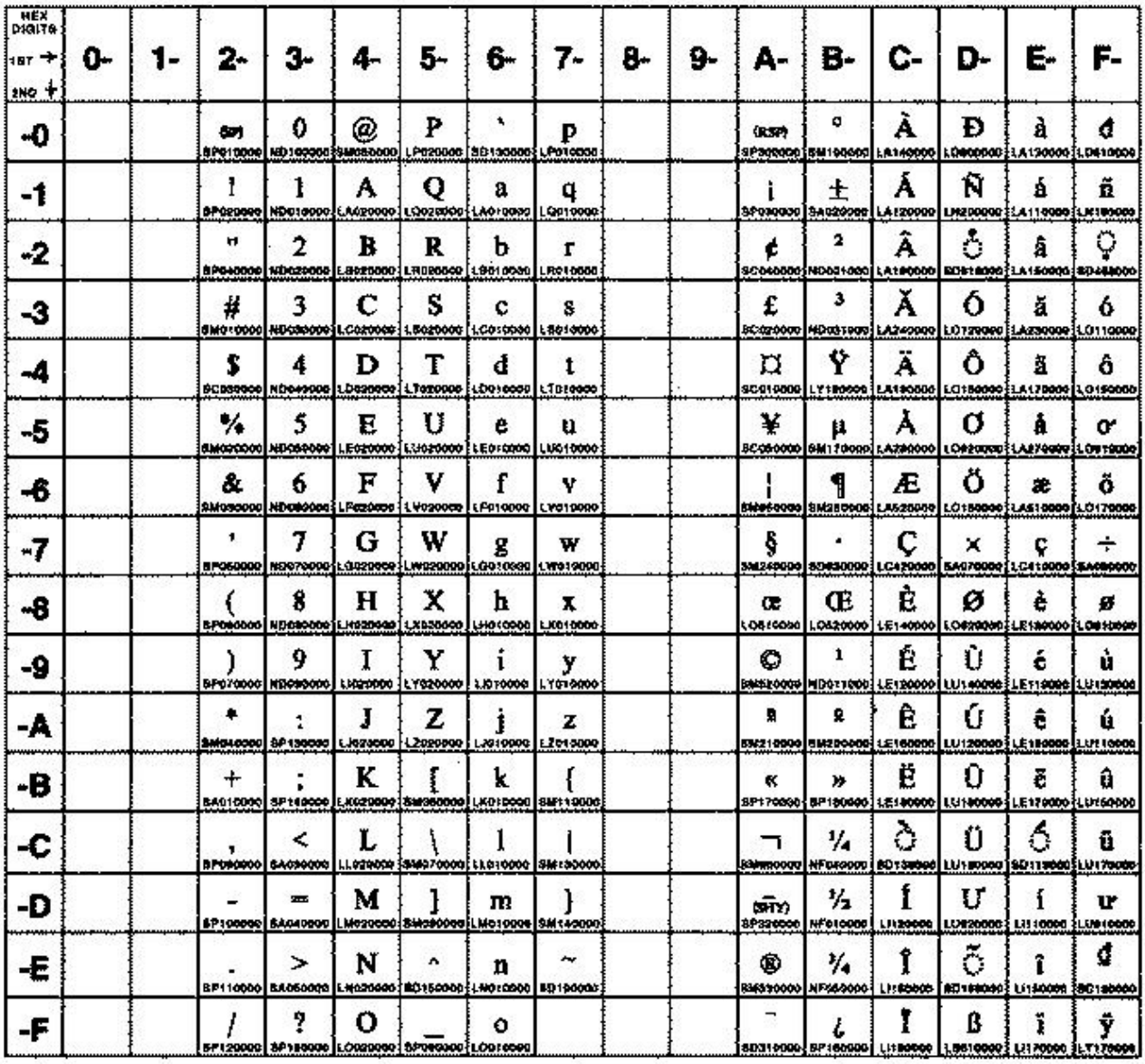
The following figure summarizes the available symbols and shows the layout of Code Set IBM-1124. For a textual representation of this code set, see “IBM-1124” on page 974.



HEX DIGITS 1ST → 2ND ↓	0-	1-	2-	3-	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0			(SP)	0	@	P	`	p			(RSP)	А	Р	а	р	№
-1			!	1	A	Q	a	q			Ё	Б	С	б	с	ё
-2			"	2	B	R	b	r			Ђ	В	Т	в	т	ђ
-3			#	3	C	S	c	s			Ѓ	Г	У	г	у	ѓ
-4			$	4	D	T	d	t			Є	Д	Ф	д	ф	є
-5			%	5	E	U	e	u			S	Е	Х	е	х	s
-6			&	6	F	V	f	v			І	Ж	Ц	ж	ц	і
-7			'	7	G	W	g	w			Ї	З	Ч	з	ч	ї
-8			(	8	H	X	h	x			Ј	И	Ш	и	ш	ј
-9			)	9	I	Y	i	y			Љ	Й	Щ	й	щ	љ
-A			*	:	J	Z	j	z			Њ	К	Ъ	к	ъ	њ
-B			+	;	K	[	k	{			Ћ	Л	Ы	л	ы	ћ
-C			,	<	L	\	l	|		Ќ	М	Ь	м	ь	ќ
-D			-	=	M	]	m	}			(SHY)	Н	Э	н	э	§
-E			.	>	N	^	n	~			Ў	О	Ю	о	ю	ў
-F			/	?	O	_	o				Џ	П	Я	п	я	џ

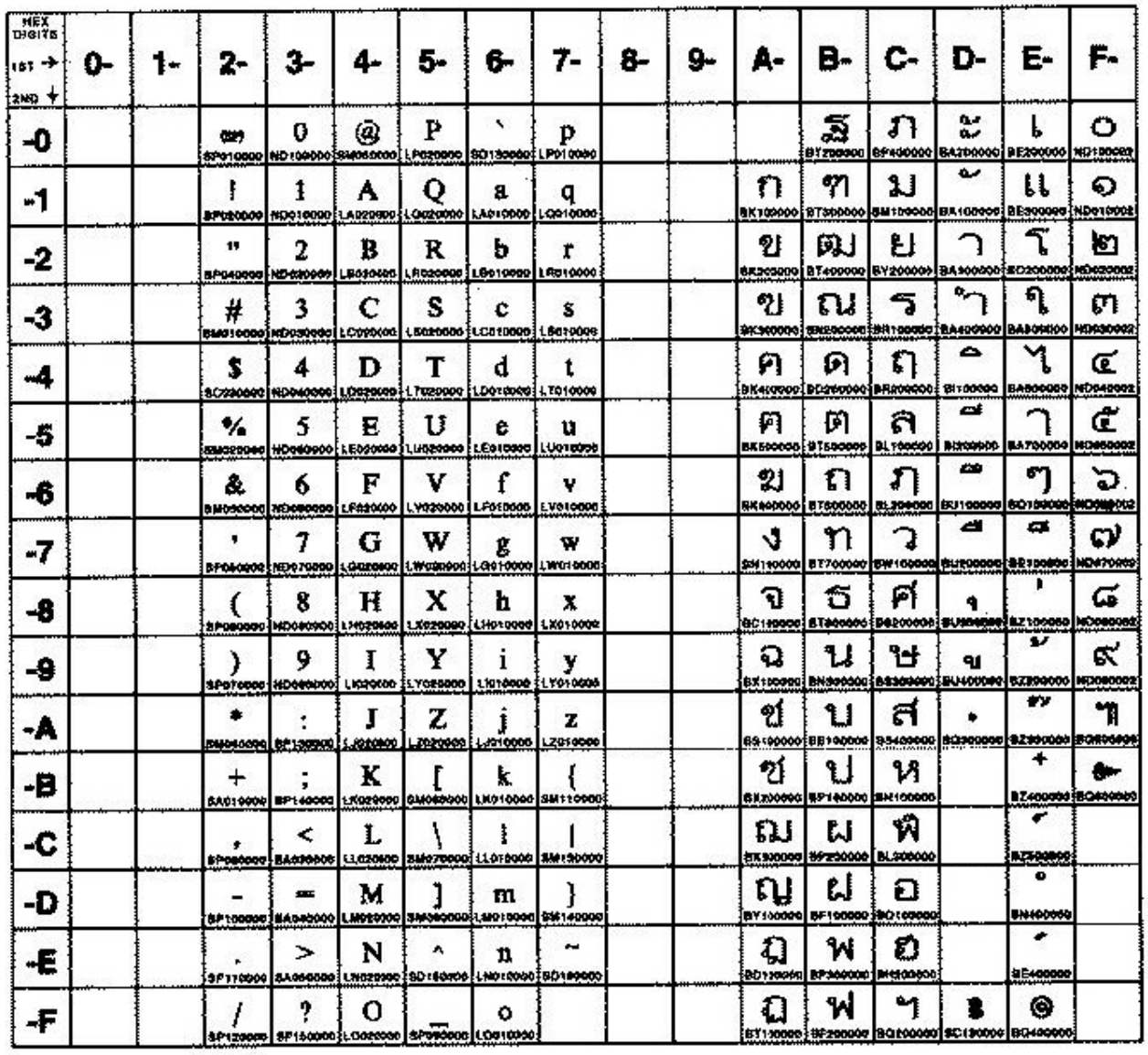
IBM-1129

The following figure summarizes the available symbols and shows the layout of Code Set IBM-1129. For a textual representation of this code set, see “IBM-1129” on page 977.



TIS-620

The following figure summarizes the available symbols and shows the layout of Code Set TIS-620. For a textual representation of this code set, see “TIS-620” on page 979.



UCS-2 and UTF-8

AIX provides a set of codesets that address the needs of a particular language or a language group. None of the codesets represented in the ISO8859 family of codesets, the PC codesets, nor the Extended Unix Code (EUC) codesets allow the mixing of characters from different scripts. With ISO8859-1, you can mix and represent the Latin 1 characters (languages principally spoken in the U.S., Canada, Western Europe, and Latin America). ISO8859-2 covers Eastern European languages; ISO8859-5 covers Cyrillic, ISO8859-6 covers Arabic, ISO8859-7 covers Greek, ISO8859-8 covers Hebrew, ISO8859-9 covers Turkish, IBM-eucJP covers Japanese, IBM-eucKR covers Korean, IBM-eucTW covers Simplified Chinese. The point is that none of the above codesets covers all of the languages.

The International Organization for Standardization (ISO) has addressed the limited language coverage by codesets by adopting Unicode as the encoding for the 2-octet form of the ISO10646 Universal Multiple-Octet Coded Character Set (UCS-2). The 32-bit form of ISO10646 is known as UCS-4 for 4-octet form. AIX has adopted the 16-bit form of ISO10646 and uses the standard label ″UCS-2″ to describe this encoding.

Although UCS-2 is ideal for an internal process code, it is not suitable for encoding plain text on traditional byte-oriented systems, such as AIX. Therefore, the external file code is X/Open's File System Safe UCS Transformation Format (FSS-UTF). This transformation format encoding is also known as UTF-8, and ″UTF-8″ is the label that is used for this encoding on AIX.

ISO10646 UCS-2 (Unicode)

Universal Coded Character Set (UCS) is the name of the ISO10646 standard that defines a single code for the representation, interchange, processing, storage, entry, and presentation of the written form of all the major languages of the world.

UCS has the following key objectives:

- Improve interoperability between systems in an open environment.
- Simplify development of internationalized products.
- Provide a base for multilingual applications.
- Make more characters available.

ISO10646 defines canonical character codes with a length of 32 bits. This provides code numbers for over 4 billion characters. When used in canonical form to represent text, the coding is referred to as UCS-4 for Universal Coded Character Set 4-byte form.

The code values from 0x0000 through 0xFFFF of ISO 10646 can be represented by a uniform character encoding of 16 bits. When used in this form to represent text, these codes are referred to as UCS-2, for Universal Character Set 2-octet form. This range is also called the Basic Multilingual Plane (BMP) of ISO10646. The standard is arranged so that the most useful characters, covering all major existing standards worldwide, are assigned within this range.

The character code values of UCS-2 are identical to those of the Unicode character encoding standard published by the Unicode Consortium.

UCS-2 defines codes for characters used in all major written languages. In addition to a set of scientific, mathematic, and publishing symbols, UCS-2 covers the following scripts:

- Arabic
- Armenian
- Bengali
- Bopomofo
- Cyrillic
- Devanagari
- Georgian
- Greek
- Gujarati
- Gurmukhi
- Hangul
- Chinese Hanzi
- Hebrew
- Hiragana
- International Phonetic Alphabet (IPA)
- Katakana
- Japanese Kanji
- Kannada
- Korean Hanja

- Laotian
- Latin
- Malayalam
- Oriya
- Tamil
- Teluga
- Thai
- Tibetan

The ability of AIX to display characters in the scripts mentioned above is limited to the availability of fonts. AIX provides bitmap fonts for most of the major languages of the world, as well as a Unicode based scalable TrueType font. Use of this font requires the TrueType font rasterizer for AIX, which is a separately installable feature.

UCS-2 encodes a number of combining characters, also known as non-spacing marks for floating diacritics. These characters are necessary in several scripts including Indic, Thai, Arabic, and Hebrew. The combining characters are used for generating characters in Latin, Cyrillic, and Greek scripts. However, the presence of combining characters creates the possibility for an alternative coding for the same text. Although the coding is unambiguous and data integrity is preserved, the processing of text that contains combining characters is more complex. To provide conformance for applications that choose not to deal with the combining characters, ISO10646 defines the following three implementation levels:

Level 1	Does not allow combining characters.
Level 2	Allows combining marks from Thai, Indic, Hebrew, and Arabic scripts.
Level 3	Allows combining marks, including ones for Latin, Cyrillic, and Greek.

UTF-8 (UCS Transformation Format)

X/Open has developed a transformation format for UCS designed for use in existing file systems. The original name of this transformation is FSS-UTF, but it is expected to be registered by ISO as UTF-8. UTF-8 is expected to become the standard transformation method, for situations where UCS is not practical. The intent is that UCS will be the process code for the transformation format, which is usable as a file code.

UTF-8 has the following properties:

- It is a superset of ASCII, in which the ASCII characters are encoded as single-byte characters with the same numeric value.
- No ASCII code values occur in multibyte characters, other than those which represent the ASCII characters.
- Conversion to and from UCS is simple and efficient.
- The first byte of a character indicates the number of bytes to follow in the multibyte character sequence and cannot occur anywhere else in the sequence.

The UTF-8 encodes UCS values in the 0 through 0x7FFFFFFF range using multibyte characters with lengths of 1, 2, 3, 4, 5, and 6 bytes. Single-byte characters are reserved for the ASCII characters in the 0 through 0x7f range. These all have the high order bit set to 0. For all character encodings of more than one byte, the initial byte determines the number of bytes used, and the high-order bit in each byte is set. Every byte that does not start with the bit combination of 10xxxxxx, where x represents a bit that may be 0 or 1, is the start of a UCS character sequence.

UTF-8 Multibyte Codes				
Bytes	**Bits**	**Hex Minimum**	**Hex Maximum**	**Byte Sequence in Binary**
1	7	00000000	0000007F	0xxxxxxx

2	11	00000080	000007FF	110xxxxx 10xxxxxx
3	16	00000800	0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
4	21	00010000	001FFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
5	26	00200000	03FFFFFF	111110xx 10xxxxxx 10xxxxxx 10xxxxx 10xxxxxx
6	31	04000000	7FFFFFFF	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

The UCS value is just the concatenation of the *x* bits in the multibyte encoding. When there are multiple ways to encode a value (for example, UCS 0), only the shortest encoding is legal.

The following subset of UTF-8 is used to encode UCS-2:

UTF-8 Multibyte Codes				
Bytes	**Bits**	**Hex Minimum**	**Hex Maximum**	**Byte Sequence in Binary**
1	7	00000000	0000007F	0xxxxxxx
2	11	00000080	000007FF	110xxxxx 10xxxxxx
3	16	00000800	0000FFFF	1110xxxx 10xxxxxx 10xxxxxx

This subset of UTF-8 requires a maximum of three (3) bytes.

Related Information

Low Function Terminal (LFT) Subsystem Overview in *AIX 5L Version 5.1 Kernel Extensions and Device Support Programming Concepts*.

- “Chapter 16. National Language Support” on page 329
- “Locale Overview for Programming” on page 330
- “Converters Overview for Programming”
- “Input Method Overview” on page 452
- Keyboard Overview

National Language Support Overview for System Management, National Language Support Overview for Devices, and Locale Overview for System Management in *AIX 5L Version 5.1 System Management Guide: Operating System and Devices*.

The **iconv** command.

Converters Overview for Programming

National Language Support (NLS) provides a base for internationalization in which data often can be changed from one code set to another. Support of several standard converters for this purpose is provided. This section discusses the following aspects of conversion:

- “Converters Introduction” on page 411
- “Standard Converters” on page 411
- “Understanding libiconv” on page 412
- “Using Converters” on page 414
- “List of Converters” on page 416

Converters Introduction

Data sent by one program to another program residing on a remote host may require conversion from the code set of the source machine to that of the receiver. For example, when communicating with a VM system, the workstation converts its ISO8859-1 data to an EBCDIC form.

Code sets define graphic characters and control character assignments to code points. These coded characters must also be converted when a program obtains data in one code set but displays it in another code set.

Two interfaces for conversions are provided:

iconv command	Allows you to request a specific conversion by naming the *FromCode* and *ToCode* code sets.
libiconv functions ("Understanding libiconv" on page 412)	Allow applications to request converters by name.

The system provides ready-to-use libraries of converters. You supply the name of the converter you want to use. The converter libraries are found in the **/usr/lib/nls/loc/iconv/*** and **/usr/lib/nls/loc/iconvTable/*** directories.

In addition to code set converters, the converter library also provides a set of network interchange converters. In a network environment, the code sets of the communications systems and the protocols of communication determine how the data should be converted.

Interchange converters are used to convert data sent from one system to another. Conversions from one internal code set to another require code set converters. When data must be converted from a sender's code set to a receiver's code set or from 8-bit data to 7-bit data, a uniform interface is required. The **iconv** subroutines provide this interface.

Standard Converters

The system supports standard converters for use with the **iconv** command and subroutines. The following list describes the different types of converters:

Code Set Converter Types	Description
Table converter ("List of PC, ISO, and EBCDIC Code Set Converters" on page 417)	Converts single-byte stateless code sets. Performs a table translation from one byte to another byte.
Algorithmic converter ("List of Multibyte Code Set Converters" on page 421)	Performs a conversion that cannot be implemented using a simple single-byte mapping table. All multibyte converters are currently implemented in this way.

Interchange Converter Types	Description
"List of Interchange Converters—7-bit" on page 425	Converts between internal code sets and ISO2022 standard interchange formats (7-bit).
"List of Interchange Converters—8-bit" on page 427	Converts between internal code sets and ISO2022 standard interchange formats (8-bit).
"List of Interchange Converters—Compound Text" on page 430	Converts between compound text and internal code sets.
"List of Interchange Converters—uucode" on page 432	Provides the same mapping as that defined in the **uuencode** and **uudecode** command.
"List of UCS-2 Interchange Converters" on page 433	Converts between UCS-2 and other code sets.
"List of UTF-8 Interchange Converters" on page 435	Converts between UTF-8 and other code sets.

Miscellaneous Converters	Description
"List of Miscellaneous Converters" on page 437	Used by some of the converters listed above.

Understanding libiconv

The **iconv** application programming interface (API) consists of three subroutines that accomplish conversion:

iconv_open	Performs the initialization required to convert characters from the code set specified by the *FromCode* parameter to the code set specified by the *ToCode* parameter. The strings specified are dependent on the converters installed in the system. If initialization is successful, the converter descriptor, **iconv_t**, is returned in its initial state.
"The iconv Subroutine" on page 440	Invokes the converter function using the descriptor obtained from the **iconv_open** subroutine. The *inbuf* parameter points to the first character in the input buffer, and the *inbytesleft* parameter indicates the number of bytes to the end of the buffer being converted. The *outbuf* parameter points to the first available byte in the output buffer, and the *outbytesleft* parameter indicates the number of available bytes to the end of the buffer. For state-dependent encoding, the subroutine is placed in its initial state by a call for which the *inbuf* value is a null pointer. Subsequent calls with the *inbuf* parameter as something other than a null pointer cause the internal state of the function to be altered as necessary.
iconv_close	Closes the conversion descriptor specified by the *cd* variable and makes it usable again.

In a network environment, two factors determine how data should be converted:

- Code sets of the sender and the receiver
- Communication protocol (8-bit or 7-bit data)

The following table outlines the conversion methods and recommends how you should convert data in different situations. See the "List of Interchange Converters—7-bit" on page 425 and the "List of Interchange Converters—8-bit" on page 427 for more information.

Outline of Methods and Recommended Choices				
	Communication with system using the same code set		Communication with system using different code set or receiver's code set is unknown	
	Protocol		Protocol	
Method to choose	7-bit only	8-bit	7-bit only	8-bit
as is	Not valid	Best choice	Not valid	Not valid if remote code set is unknown
fold7	OK	OK	Best choice	OK
fold8	Not valid	OK	Not valid	Best choice
uucode	Best choice	OK	Not valid	Not valid

If the sender uses the same code set as the receiver, there are two possibilities:

- When protocol allows 8-bit data, the data can be sent without conversions.
- When protocol allows only 7-bit data, the 8-bit code points must be mapped to 7-bit values. Use the iconv interface and one of the following methods:

"List of Interchange Converters—uucode" on page 432	Provides the same mapping as the **uuencode** and **uudecode** commands. This is the recommended method.

"List of Interchange Converters—7-bit" on page 425	Converts internal code sets using 7-bit data. This method passes ASCII without any change.

If the sender uses a code set different from the receiver, there are two possibilities:

- When protocol allows only 7-bit data, use the fold7 method.
- When protocol allows 8-bit data and you know the receiver's code set, use the iconv interface to convert the data. If you do not know the receiver's code set, use the following method:

"List of Interchange Converters—8-bit" on page 427	Converts internal code sets to standard interchange formats. The 8-bit data is transmitted and the information is preserved so that the receiver can reconstruct the data in its code set.

Using the iconv_open Subroutine

The following examples illustrate how to use the **iconv_open** subroutine in different situations:

- Sender and receiver use the same code sets:

  If the protocol allows 8-bit data, you can send data without converting it.

  If the protocol allows only 7-bit data, do the following:

```
Sender:
 cd = iconv_open("uucode", nl_langinfo(CODESET));


Receiver:
 cd = iconv_open(nl_langinfo(CODESET), "uucode");
```

- Sender and receiver use different code sets:

  If the protocol allows 8-bit data and the receiver's code set is unknown, do the following:

```
Sender:
 cd = iconv_open("fold8", nl_langinfo(CODESET));


Receiver:
 cd = iconv_open(nl_langinfo(CODESET),"fold8" );
```

  If the protocol allows only 7-bit data, do the following:

```
Sender:
 cd = iconv_open("fold7", nl_langinfo(CODESET));


Receiver:
 cd = iconv_open(nl_langinfo(CODESET), "fold7" );
```

How the iconv_open Subroutine Finds Converters

The **iconv_open** subroutine uses the **LOCPATH** environment variable to search for a converter whose name is in the form:

iconv/*FromCodeSet_ToCodeSet*

The *FromCodeSet* string represents the sender's code set, and the *ToCodeSet* string represents the receiver's code set. The underscore character separates the two strings.

> **Note:** All **setuid** and **setgid** programs will ignore the **LOCPATH** environment variable.

Since the iconv converter is aloadable object module, a different object is required when running in the 64-bit environment. In the 64-bit environment, the **iconv**_open routine will use the LOCPATH environment variable to search for a converter whose name is in the form:

```
iconv/FromCodeSet_ToCodeSet__64.
```

The iconv library will automatically choose whether to load the standard converter object or the 64-bit converter object.

If the **iconv_open** subroutine does not find the converter, it uses the *from,to* pair to search for a file that defines a table-driven conversion. The file contains a conversion table created by the **genxlt** command.

The iconvTable converter uses the **LOCPATH** environment variable to search for a file whose name is in the form:

iconvTable/*FromCodeSet_ToCodeSet*

If the converter is found, it performs a load operation and is initialized. The converter descriptor, **iconv_t**, is returned in its initial state.

Converter Programs versus Tables

Converter programs are executable functions that convert data according to a set of rules. Converter tables are single-byte conversion tables that perform stateless conversions. Programs and tables are in separate directories:

/usr/lib/nls/loc/iconv	Converter programs
/usr/lib/nls/loc/iconvTable	Converter tables.

After a converter program is compiled and linked with the **libiconv.a** library, the program is placed in the **/usr/lib/nls/loc/iconv** directory.

To build a table converter, build a source converter table file. Use the **genxlt** command to compile translation tables into a format understood by the table converter. The output file is then placed in the **/usr/lib/nls/loc/iconvTable** directory.

Unicode and Universal Converters

Unicode (or UCS-2) conversion tables are found in:

```
$LOCPATH/uconvTable/*CodeSet*
```

The **$LOCPATH/uconv/UCSTBL** converter program is used to perform the conversion to and from UCS-2 using the **iconv** utilities. For the **iconv** utilities to use these **uconvTable** conversion tables, links must be set up within the **$LOCPATH/iconv** directory, for example, for code set ″X.″

```
ln -s /usr/lib/nls/loc/uconv/UCSTBL /usr/lib/nls/loc/iconv/X_UCS-2
ln -s /usr/lib/nls/loc/uconv/UCSTBL /usr/lib/nls/loc/iconv/UCS-2_X
```

A ″Universal converter″ program is provided that can be used to convert between any two code sets whose conversions to and from UCS-2 is defined. Given the following uconvTables:

```
X     -> UCS-2
UCS-2 -> Y
```

a universal conversion can be defined that maps

```
X -> UCS-2 -> Y
```

by use of the **$LOCPATH/iconv/Universal_UCS_Conv**. The conversion X->Y is set by defining links to the universal converter, for example:

```
ln-s /usr/lib/nls/loc/iconv/Universal_UCS_Conv /usr/lib/nls/loc/iconv/X_Y
```

Using Converters

The iconv interface is a set of subroutines used to open, perform, and close conversions:

- **iconv_open**
- **iconv** (“The iconv Subroutine” on page 440)

- **iconv_close**

Code Set Conversion Filter Example

The following example shows how you can use these subroutines to create a code set conversion filter that accepts the *ToCode* and *FromCode* parameters as input arguments:

```
#include <stdio.h>
#include <nl_types.h>
#include <iconv.h>
#include <string.h>
#include <errno.h>
#include <locale.h>

#define ICONV_DONE() (r>=0)
#define ICONV_INVAL() (r<0) && (errno==EILSEQ))
#define ICONV_OVER() (r<0) && (errno==E2BIG))
#define ICONV_TRUNC() (r<0) && (errno==EINVAL))

#define USAGE 1
#define ERROR 2
#define INCOMP 3

char ibuf[BUFSIZ], obuf[BUFSIZ];

extern int errno;

main (argc,argv)
int argc;
char **argv;
{
 size_t  ileft,oleft;
 nl_catd catd;
 iconv_t cd;
 int r;
 char *ip,*op;

 setlocale(LC_ALL,"");
 catd = catopen (argv[0],0);

 if(argc!=3){
  fprintf(stderr,
   catgets (catd,NL_SETD,USAGE,"usage;conv fromcode tocode\n"));
  exit(1);
 }

 cd=iconv_open(argv[2],argv[1]);

ileft=0;

while(!feof(stdin)) {
 /*
 * After the next operation,ibuf will
 * contain new data plus any truncated
 * data left from the previous read.
 */
 ileft+=fread(ibuf+ileft,1,BUFSIZ-ileft,stdin);
 do {
  ip=ibuf;
  op=obuf;
  oleft=BUFSIZ;

  r=iconv(cd,&ip,&ileft,&op,&oleft);

  if(ICONV_INVAL()){
   fprintf(stderr,
      catgets(catd,NL_SETD,ERROR,"invalid input\n"));
```

```
   exit(2);
 }

 fwrite(obuf,1,BUFSIZ-oleft,stdout);

 if(ICONV_TRUNC() || ICONV_OVER())
  /*
  *Data remaining in buffer-copy
  *it to the beginning
  */

  memcpy(ibuf,ip,ileft);

  /*
  *loop until all characters in the input
  *buffer have been converted.
  */
 } while(ICONV_OVER());
}

 if(ileft!=0){
  /*
  *This can only happen if the last call
  *to iconv() returned ICONV_TRUNC, meaning
  *the last data in the input stream was
  *incomplete.
  */
 fprintf(stderr,catgets(catd,NL_SETD,INCOMP,"input incomplete\n"));
 exit(3);
 }

 iconv_close(cd);
 exit(0);
}
```

Naming Converters

Code set names are in the form *CodesetRegistry*-*CodesetEncoding* where:

CodesetRegistry	Identifies the registration authority for the encoding. The *CodesetRegistry* must be made of characters from the portable code set (usually A-Z and 0-9).
CodesetEncoding	Identifies the coded character set defined by the registered authority.

The *from,to* variable used by the **iconv** command and **iconv_open** subroutine identifies a file whose name should be in the form */usr/lib/nls/loc/iconv/%f_%t* or */usr/lib/nls/loc/iconvTable/%f_%t*, where:

%f	Represents the *FromCode* set name.
%t	Represents the *ToCode* set name.

List of Converters

Converters change data from one code set to another. The sets of converters supported with the ICONV library are in the following sections. All converters shipped with the BOS Runtime Environment are located in the **/usr/lib/nls/loc/iconv/*** or **/usr/lib/nls/loc/iconvTable/*** directory.

These directories also contain *private* converters; that is, they are used by other converters. However, users and programs should only depend on the converters in the following lists.

Any converter shipped with the BOS Runtime Environment and not listed here should be considered private and subject to change or deletion. Converters supplied by other products can be placed in the **/usr/lib/nls/loc/iconv/*** or **/usr/lib/nls/loc/iconvTable/*** directory.

Programmers are encouraged to use registered code set names or code set names associated with an application. The X Consortium maintains a registry of code set names for reference. See the “Code Set Overview” on page 379 for more information about code sets.

List of PC, ISO, and EBCDIC Code Set Converters

These converters provide conversion between PC, ISO, and EBCDIC single-byte stateless code sets. The following types of conversions are supported: PC to/from ISO, PC to/from EBCDIC, and ISO to/from EBCDIC.

Conversion is provided between compatible code sets such as Latin-1 to Latin-1 and Greek to Greek. However, conversion between different EBCDIC national code sets is not supported. For information about converting between incompatible character sets refer to the “List of Interchange Converters—7-bit” on page 425 and the “List of Interchange Converters—8-bit” on page 427.

Conversion tables in the **iconvTable** directory are created by the **genxlt** command.

Compatible Code Set Names: The following table lists code set names that are compatible. Each line defines to/from strings that may be used when requesting a converter.

Note: The PC and ISO code sets are ASCII-based.

Code Set Compatibility				
Character Set	**Languages**	**PC**	**ISO**	**EBCDIC**
Latin-1	U.S. English, Portuguese, Canadian French	IBM-850	ISO8859-1	IBM-037
Latin-1	Danish, Norwegian	IBM-850	ISO8859-1	IBM-277
Latin-1	Finnish, Swedish	IBM-850	ISO8859-1	IBM-278
Latin-1	Italian	IBM-850	ISO8859-1	IBM-280
Latin-1	Japanese	IBM-850	ISO8859-1	IBM-281
Latin-1	Spanish	IBM-850	ISO8859-1	IBM-284
Latin-1	U.K. English	IBM-850	ISO8859-1	IBM-285
Latin-1	German	IBM-850	ISO8859-1	IBM-273
Latin-1	French	IBM-850	ISO8859-1	IBM-297
Latin-1	Belgian, Swiss German	IBM-850	ISO8859-1	IBM-500

Latin-2	Croatian, Czechoslovakian, Hungarian, Polish, Romanian, Serbian Latin, Slovak, Slovene	IBM-852	ISO88859-2	IBM-870
Cyrillic	Bulgarian, Macedonian, Serbian Cyrillic, Russian	IBM-855	ISO8859-5	IBM-880 IBM-1025
Cyrillic	Russian	IBM-866	ISO8859-5	IBM-1025
Hebrew	Hebrew	IBM-856 IBM-862	ISO8859-8	IBM-424 IBM-803
Turkish	Turkish	IBM-857	ISO8859-9	IBM-1026
Arabic	Arabic	IBM-864 IBM-1046	ISO8859-6	IBM-420
Greek	Greek	IBM-869	ISO8859-7	IBM-875
Greek	Greek	IBM-869	ISO8859-7	IBM-875
Baltic	Lithuanian, Latvian, Estonian	IBM-921 IBM-922		IBM-1112 IBM-1122

Note: A character that exists in the source code set but does not exist in the target code set is converted to a converter-defined substitute character.

Files: The following table describes the **inconvTable** converters found in the **/usr/lib/nls/loc/iconvTable** directory:

iconvTable Converters		
Converter Table	**Description**	**Language**
IBM-037_IBM-850	IBM-037 to IBM-850	U.S. English, Portuguese, Canadian-French
IBM-273_IBM-850	IBM-273 to IBM-850	German
IBM-277_IBM-850	IBM-277 to IBM-850	Danish, Norwegian
IBM-278_IBM-850	IBM-278 to IBM-850	Finnish, Swedish
IBM-280_IBM-850	IBM-280 to IBM-850	Italian
IBM-281_IBM-850	IBM-281 to IBM-850	Japanese-Latin
IBM-284_IBM-850	IBM-284 to IBM-850	Spanish
IBM-285_IBM-850	IBM-285 to IBM-850	U.K. English
IBM-297_IBM-850	IBM-297 to IBM-850	French
IBM-420_IBM_1046	IBM-420 to IBM-1046	Arabic
IBM-424_IBM-856	IBM-424 to IBM-856	Hebrew
IBM-424_IBM-862	IBM-424 to IBM-862	Hebrew
IBM-500_IBM-850	IBM-500 to IBM-850	Belgian, Swiss German
IBM-803_IBM-856	IBM-803 to IBM-856	Hebrew
IBM-803_IBM-862	IBM-803 to IBM-862	Hebrew
IBM-850_IBM-037	IBM-850 to IBM-037	U.S. English, Portuguese, Canadian-French
IBM-850_IBM-273	IBM-850 to IBM-273	German
IBM-850_IBM-277	IBM-850 to IBM-277	Danish, Norwegian
IBM-850_IBM-278	IBM-850 to IBM-278	Finnish, Swedish

IBM-850_IBM-280	IBM-850 to IBM-280	Italian
IBM-850_IBM-281	IBM-850 to IBM-281	Japanese-Latin
IBM-850_IBM-284	IBM-850 to IBM-284	Spanish
IBM-850_IBM-285	IBM-850 to IBM-285	U.K. English
IBM-850_IBM-297	IBM-850 to IBM-297	French
IBM-850_IBM-500	IBM-850 to IBM-500	Belgian, Swiss German
IBM-856_IBM-424	IBM-856 to IBM-424	Hebrew
IBM-856_IBM-803	IBM-856 to IBM-803	Hebrew
IBM-856_IBM-862	IBM-856 to IBM-862	Hebrew
IBM-862_IBM-424	IBM-862 to IBM-424	Hebrew
IBM-862_IBM-803	IBM-862 to IBM-803	Hebrew
IBM-862_IBM-856	IBM-862 to IBM-856	Hebrew
IBM-864_IBM-1046	IBM-864 to IBM-1046	Arabic
IBM-921_IBM-1112	IBM-921 to IBM-1112	Lithuanian, Latvian
IBM-922_IBM-1122	IBM-922 to IBM-1122	Estonian
IBM-1112_IBM-921	IBM-1121 to IBM-921	Lithuanian, Latvian
IBM-1122_IBM-922	IBM-1122 to IBM-922	Estonian
IBM-1046_IBM-420	IBM-1046 to IBM-420	Arabic
IBM-1046_IBM-864	IBM-1046 to IBM-864	Arabic
IBM-037_ISO8859-1	IBM-037 to ISO8859-1	U.S. English, Portuguese, Canadian French
IBM-273_ISO8859-1	IBM-273 to ISO8859-1	German
IBM-277_ISO8859-1	IBM-277 to ISO8859-1	Danish, Norwegian
IBM-278_ISO8859-1	IBM-278 to ISO8859-1	Finnish, Swedish
IBM-280_ISO8859-1	IBM-280 to ISO8859-1	Italian
IBM-281_ISO8859-1	IBM-281 to ISO8859-1	Japanese-Latin
IBM-284_ISO8859-1	IBM-284 to ISO8859-1	Spanish
IBM-285_ISO8859-1	IBM-285 to ISO8859-1	U.K. English
IBM-297_ISO8859-1	IBM-297 to ISO8859-1	French
IBM-420_ISO8859-6	IBM-420 to ISO8859-6	Arabic
IBM-424_ISO8859-8	IBM-424 to ISO8859-8	Hebrew
IBM-500_ISO8859-1	IBM-500 to ISO8859-1	Belgian, Swiss German
IBM-803_ISO8859-8	IBM-803 to ISO8859-8	Hebrew
IBM-852_ISO8859-2	IBM-852 to ISO8859-2	Croatian, Czechoslovakian, Hungarian, Polish, Romanian, Serbian Latin, Slovak, Slovene
IBM-855_ISO8859-5	IBM-855 to ISO8859-5	Bulgarian, Macedonian, Serbian Cyrillic, Russian
IBM-866_ISO8859-5	IBM-866 to ISO8859-5	Russian
IBM-869_ISO8859-7	IBM-869 to ISO8859-7	Greek
IBM-875_ISO8859-7	IBM-875 to ISO8859-7	Greek

IBM-870_ISO8859-2	IBM-870 to ISO8859-2	Croatian, Czechoslovakian, Hungarian, Polish, Romanian, Serbian, Slovak, Slovene
IBM-880_ISO8859-5	IBM-880 to ISO8859-5	Bulgarian, Macedonian, Serbian Cyrillic, Russian
IBM-1025_ISO8859-5	IBM-1025 to ISO8859-5	Bulgarian, Macedonian, Serbian Cyrillic, Russian
IBM-857_ISO8859-9	IBM-857 to ISO8859-9	Turkish
IBM-1026_ISO8859-9	IBM-1026 to ISO8859-9	Turkish
IBM-850_ISO8859-1	IBM-850 to ISO8859-1	Latin
IBM-856_ISO8859-8	IBM-856 to ISO8859-8	Hebrew
IBM-862_ISO8859-8	IBM-862 to ISO8859-8	Hebrew
IBM-864_ISO8859-6	IBM-864 to ISO8859-6	Arabic
IBM-1046_ISO8859-6	IBM-1046 to ISO8859-6	Arabic
ISO8859-1_IBM-850	ISO8859-1 to IBM-850	Latin
ISO8859-6_IBM-864	ISO8859-6 to IBM-864	Arabic
ISO8859-6_IBM-1046	ISO8859-6 to IBM-1046	Arabic
ISO8859-8_IBM-856	ISO8859-8 to IBM-856	Hebrew
ISO8859-8_IBM-862	ISO8859-8 to IBM-862	Hebrew
ISO8859-1_IBM-037	ISO8859-1 to IBM-037	U.S. English, Portuguese, Canadian French
ISO8859-1_IBM-273	ISO8859-1 to IBM-273	German
ISO8859-1_IBM-277	ISO8859-1 to IBM-277	Danish, Norwegian
ISO8859-1_IBM-278	ISO8859-1 to IBM-278	Finnish, Swedish
ISO8859-1_IBM-280	ISO8859-1 to IBM-280	Italian
ISO8859-1_IBM-281	ISO8859-1 to IBM-281	Japanese-Latin
ISO8859-1_IBM-284	ISO8859-1 to IBM-284	Spanish
ISO8859-1_IBM-285	ISO8859-1 to IBM-285	U.K. English
ISO8859-1_IBM-297	ISO8859-1 to IBM-297	French
ISO8859-1_IBM-500	ISO8859-1 to IBM-500	Belgian, Swiss German
ISO8859-2_IBM-852	ISO8859-2 to IBM-852	Croatian, Czechoslovakian, Hungarian, Polish, Romanian, Serbian Latin, Slovak, Slovene
ISO8859-2_IBM-870	ISO8859-2 to IBM-870	Croatian, Czechoslovakian, Hungarian, Polish, Romanian, Serbian Latin, Slovak, Slovene
ISO8859-5_IBM-855	ISO8859-5 to IBM-855	Bulgarian, Macedonian, Serbian Cyrillic, Russian
ISO8859-5_IBM-880	ISO8859-5 to IBM-880	Bulgarian, Macedonian, Serbian Cyrillic, Russian
ISO8859-5_IBM-1025	ISO8859-5 to IBM-1025	Bulgarian, Macedonian, Serbian Cyrillic, Russian
ISO8859-6_IBM-420	ISO8859-6 to IBM-420	Arabic
ISO8859-5_IBM-866	ISO8859-5 to IBM-866	Russian
ISO8859-7_IBM-869	ISO8859-7 to IBM-869	Greek

ISO8859-7_IBM-875	ISO8859-7 to IBM-875	Greek
ISO8859-8_IBM-424	ISO8859-8 to IBM-424	Hebrew
ISO8859-8_IBM-803	ISO8859-8 to IBM-803	Hebrew
ISO8859-9_IBM-857	ISO8859-9 to IBM-857	Turkish
ISO8859-9_IBM-1026	ISO8859-9 to IBM-1026	Turkish

List of Multibyte Code Set Converters

Multibyte code-set converters convert characters among the following code-sets:

- PC multibyte code sets
- EUC multibyte code sets (ISO-based)
- EBCDIC multibyte code sets

The following table lists code set names that are compatible. Each line defines to/from strings that may be used when requesting a converter.

Code Set Compatibility			
Language	**PC**	**ISO**	**EBCDIC**
Japanese	IBM-932	IBM-eucJP	IBM-930, IBM-939
Japanese (MS compatible)	IBM-943	IBM-eucJP	IBM-930, IBM-939
Korean	IBM-934	IBM-eucKR	IBM-933
Traditional Chinese	IBM-938, big-5	IBM-eucTW	IBM-937
Simplified Chinese	IBM-1381	IBM-eucCN	IBM-935

1. Conversions between Simplified and Traditional Chinese are provided (IBM-eucTW <—> IBM-eucCN and big5 <—> IBM-eucCN).
2. UTF-8 is an additional code set. See “List of UTF-8 Interchange Converters” on page 435 for more information.

Files: The following list describes the Multibyte Code Set converters that are found in the **/usr/lib/nls/loc/iconv** directory.

Converter	**Description**
IBM-eucJP_IBM-932	IBM-eucJP to IBM-932
IBM-eucJP_IBM-943	IBM-eucJP to IBM-943
IBM-eucJP_IBM-930	IBM-eucJP to IBM-930
IBM-eucCN_IBM-936(PC5550)	IBM-eucCN to IBM-936(PC5550)
IBM-eucCN_IBM-935	IBM-eucCN to IBM-935
IBM-eucJP_IBM-939	IBM-eucJP to IBM-939
IBM-eucCN_IBM-1381	IBM-eucCN to IBM-1381
IBM-943_IBM-932	IBM-943 to IBM-932
IBM-932_IBM-943	IBM-932 to IBM-943
IBM-930_IBM-932	IBM-930 to IBM-932
IBM-930_IBM-943	IBM-930 to IBM-943
IBM-930_IBM-eucJP	IBM-930 to IBM-eucJP

Converter	Description
IBM-932_IBM-eucJP	IBM-932 to IBM-eucJP
IBM-932_IBM-930	IBM-932 to IBM-930
IBM-943_IBM-eucJP	IBM-943 to IBM-eucJP
IBM-943_IBM-930	IBM-943 to IBM-930
IBM-936(PC5550)_IBM-935	IBM-936(PC5550) to IBM-935
IBM-936_IBM-935	IBM-936 to IBM-935
IBM-932_IBM-939	IBM-932 to IBM-939
IBM-939_IBM-932	IBM-939 to IBM-932
IBM-943_IBM-939	IBM-943 to IBM-939
IBM-939_IBM-943	IBM-939 to IBM-943
IBM-935_IBM-936(PC5550)	IBM-935 to IBM-936(PC5550)
IBM-935_IBM-936	IBM-935 to IBM-936
IBM-1381_IBM-935	IBM-1381 to IBM-935
IBM-935_IBM-1381	IBM-935 to IBM-1381
IBM-935_IBM-eucCN	IBM-935 to IBM-eucCN
IBM-936(PC5550)_IBM-eucCN	IBM-936(PC5550) to IBM-eucCN
IBM-eucTW_IBM-eucCN	IBM-eucTW to IBM-eucCN
big5_IBM-eucCN	big5 to IBM-eucCN
IBM-1381_IBM-eucCN	IBM-1381 to IBM-eucCN
IBM-939_IBM-eucJP	IBM-939 to IBM-eucJP
IBM-eucKR_IBM-934	IBM-eucKR to IBM-934
IBM-934_IBM-eucKR	IBM-934 to IBM-eucKR
IBM-eucKR_IBM-933	IBM-eucKR to IBM-933
IBM-933_IBM-eucKR	IBM-933 to IBM-eucKR
IBM-eucTW_IBM-937	IBM-eucTW to IBM-937
IBM-938_IBM-937	IBM-938 to IBM-937
big-5_IBM-937	big-5 to IBM-937
IBM-eucCN_IBM-eucTW	IBM-eucCN to IBM-eucTW
IBM-937_IBM-eucTW	IBM-937 to IBM-eucTW
IBM-937_IBM-938	IBM-937 to IBM-938
IBM-eucTW_IBM-938	IBM_eucTW to IBM_938
IBM-eucCN_big5	IBM-eucCN to big5
IBM-eucTW_big-5	IBM_eucTW to big-5
IBM-937_big-5	IBM-937 to big-5
CNS11643.1992-3_IBM-eucTW	CNS11643.1992-3 to IBM_eucTW
CNS11643.1992-3-GL_IBM-eucTW	CNS11643.1992-3-GL to IBM_eucTW
CNS11643.1992-3-GR_IBM-eucTW	CNS11643.1992-3-GR to IBM_eucTW
CNS11643.1992-4_IBM-eucTW	CNS11643.1992-4 to IBM_eucTW
CNS11643.1992-4-GL_IBM-eucTW	CNS11643.1992-4-GL to IBM_eucTW
CNS11643.1992-4-GR_IBM-eucTW	CNS11643.1992-4-GR to IBM_eucTW
IBM-eucTW_CNS11643.1992-3	IBM_eucTW to CNS11643.1992-3

Converter	Description
IBM-eucTW_CNS11643.1992-3-GL	IBM_eucTW to CNS11643.1992-3-GL
IBM-eucTW_CNS11643.1992-3-GR	IBM_eucTW to CNS11643.1992-3-GR
IBM-eucTW_CNS11643.1992-4	IBM_eucTW to CNS11643.1992-4
IBM-eucTW_CNS11643.1992-4-GL	IBM_eucTW to CNS11643.1992-4-GL
IBM-eucTW_CNS11643.1992-4-GR	IBM_eucTW to CNS11643.1992-4-GR
IBM-eucCN_GB2312.1980-1	IBM-eucCN to GB2312.1980-1
IBM-eucCN_GB2312.1980-1-GL	IBM-eucCN to GB2312.1980-1-GL
IBM-eucCN_GB2312.1980-1-GR	IBM-eucCN to GB2312.1980-1-GR
IBM-937_csic	IBM-937 to csic
csic_IBM-937	csic to IBM-937
IBM-938_csic	IBM-938 to csic
csic_IBM-938	csic to IBM-938
IBM-eucTW_ccdc	IBM-eucTW to ccdc
ccdc_IBM-eucTW	ccdc to IBM-eucTW
IBM-eucTW_cns	IBM-eucTW to cns
cns_IBM-eucTW	cnd to IBM-eucTW
IBM-eucTW_csic	IBM-eucTW to csic
csic_IBM-eucTW	csic to IBM-eucTW
IBM-eucTW_sops	IBM-ecuTW to sops
sops_IBM-eucTW	sops to IBM-eucTW
IBM-eucTW_tca	IBM-eucTW to tca
tca_IBM-eucTW	tca to IBM-eucTW
big5_cns	big5 to cns
cns_big5	cns to big5
big5_csic	big5 to csic
csic_big5	csic to big5
big5_ttc	big5 to ttc
ttc_big5	ttc to big5
big5_ttcmin	big5 to ttcmin
ttcmin_big5	ttcmin to big5
big5_unicode	big5 to unicode
unicode_big5	unicode to big5
big5_wang	big5 to wang
wang_big5	wang to big5
ccdc_csic	ccdc to csic
csic_ccdc	csic to_ccdc
csic_sops	csic to sops
sops_csic	sops to csic
CNS11643.1986-1_big5	CNS11643.1986-1 to big5
big5_CNS11643.1986-1	big5 to CNS11643.1986-1
CNS11643.1986-1-GR_big5	CNS11643.1986-1-GR to big5

Converter	Description
big5_CNS11643.1986-1-GR	big5 to CNS11643.1986-1-GR
CNS11643.1986-2_big5	CNS11643.1986-2 to big5
big5_CNS11643.1986-2	big5 to CNS11643.1986-2
CNS11643.1986-2-GR_big5	CNS11643.1986-2-GR to big5
big5_CNS11643.1986-2-GR	big5 to CNS11643.1986-2-GR
CNS11643.CT-GR_big5	CNS11643.CT-GR to big5
big5_CNS11643.CT-GR	big5 to CNS11643.CT-GR
IBM-sbdTW-GR_big5	IBM-sbdTW-GR to big5
big5_IBM-sbdTW-GR	big5 to IBM-sbdTW-GR
IBM-sbdTW.CT-GR_big5	IBM-sbdTW.CT-GR to big5
big5_IBM-sbdTW.CT-GR	big5 to IBM-sbdTW.CT-GR
IBM-sbdTW_big5	IBM-sbdTW to big5
big5_IBM-sbdTW	big5 to IBM-sbdTW
IBM-udcTW-GR_big5	IBM-udcTW-GR to big5
big5_IBM-udcTW-GR	big5 to IBM-udcTW-GR
IBM-udcTW.CT-GR_big5	IBM-udcTW.CT-GR to big5
big5_IBM-udcTW.CT-GR	big5 to IBM-udcTW.CT-GR
ISO8859-1_big5	ISO8859 to big5
big5_ISO8859-1	big5 to ISO8859
IBM-sbdTW_big5	IBM-sbdTW to big5
big5_IBM-sbdTW	big5 to IBM-sbdTW
big5_ASCII-GR	big5 to ASCII-GR
ASCII-GR_big5	ASCII-GR to big5
GBK_big5	GBK to big5
big5_GBK	big5 to GBK
GBK_IBM-eucTW	GBK to IBM-eucTW
IBM-eucTW_GBK	IBM-eucTW to GBK
CNS11643.1986-1_GBK	CNS11643.1986-1 to GBK
GBK_CNS11643.1986-1	GBK to CNS11643.1986-1
CNS11643.1986-2_GBK	CNS11643.1986-2 to GBK
GBK_CNS11643.1986-2	GBK to CNS11643.1986-2
CNS11643.1986-1-GR_GBK	CNS11643.1986-1-GR to GBK
GBK_CNS11643.1986-1-GR	GBK to CNS11643.1986-1-GR
CNS11643.1986-2-GR_GBK	CNS11643.1986-2-GR to GBK
GBK_CNS11643.1986-2-GR	GBK to CNS11643.1986-2-GR
CNS11643.1986-1-GL_GBK	CNS11643.1986-1-GL to GBK
GBK_CNS11643.1986-1-GL	GBK to CNS11643.1986-1-GL
CNS11643.1986-2-GL_GBK	CNS11643.1986-2-GL to GBK
GBK_CNS11643.1986-2-GL	GBK to CNS11643.1986-2-GL
CNS11643.CT-GR_GBK	CNS11643.CT-GR to GBK
GBK_CNS11643.CT-GR	GBK to CNS11643.CT-GR

Converter	Description
GB2312.1980.CT-GR_GBK	GB2312.1980.CT-GR to GBK
GBK_GB2312.1980.CT-GR	GBK to GB2312.1980.CT-GR
GB2312.1980-0_GBK	GBK2312.1980-0 to GBK
GBK_GB2312.1980-0	GBK to GBK2312.1980-0
GB2312.1980-0-GR_GBK	GB2312.1980-0-GR to GBK
GBK_GB2312.1980-0-GR	GBK to GB2312.1980-0-GR
GB2312.1980-0-GL_GBK	GB2312.1980-0-GL to GBK
GBK_GB2312.1980-0-GL	GBK to GB2312.1980-0-GL
ASCII-GR_GBK	ASCII-GR to GBK
GBK_ASCII-GR	GBK to ASCII-GR
ISO8859-1_GBK	ISO8859-1 to GBK
GBK_ISO8859-1	GBK to ISO8859-1
IBM-eucCN_GBK	IBM-eucCN to GBK
GBK_IBM-eucCN	GBK to IBM-eucCN

List of Interchange Converters—7-bit

This converter provides conversion between internal code and 7-bit standard interchange formats (fold7). The fold7 name identifies encodings that can be used to pass text data through 7-bit mail protocols. The encodings are based on ISO2022. For more information about fold7 see “Understanding libiconv” on page 412.

The fold7 converters convert characters from a code set to a canonical 7-bit encoding that identifies each character. This type of conversion is useful in networks where clients communicate with different code sets but use the same character sets. For example:

IBM-850 <—> ISO8859-1	Common Latin characters
IBM-932 <—>IBM-eucJP	Common Japanese characters

The following escape sequences designate standard code sets:

Escape Sequence	Standard Code Set
01/11 02/04 04/00	GL JIS X0208.1978-0.
01/11 02/04 02/08 04/01	GL left half of GB2312.1980-0.
01/11 02/08 04/02	GL 7-bit ASCII or left half of ISO8859-1.
01/11 02/14 04/01	GL right half of ISO8859-1.
01/11 02/14 04/02	GL right half of ISO8859-2.
01/11 02/14 04/03	GL right half of ISO8859-3.
01/11 02/14 04/04	GL right half of ISO8859-4.
01/11 02/14 04/06	GL right half of ISO8859-7.
01/11 02/14 04/07	GL right half of ISO8859-6.
01/11 02/14 04/08	GL right half of ISO8859-8.
01/11 02/14 04/12	GL right half of ISO8859-5.
01/11 02/14 04/13	GL right half of ISO8859-9.

Escape Sequence	Standard Code Set
01/11 02/08 04/09	GL right half of JIS X0201.1976-0.
01/11 02/08 04/10	GL left half of JIS X0201.1976.
01/11 02/04 04/02	GL JIS X0208.1983-0.
01/11 02/04 02/08 04/02	GL JIS X0208.1983-0.
01/11 02/04 02/08 04/00	GL JISX0208.1978-0.
01/11 02/05 02/15 03/01 M L 06/09 06/02 06/13 02/13 03/08 03/05 03/00 00/02	GL right half of IBM-850 unique characters. Characters common to ISO8859-1 do not use this escape sequence.
01/11 02/05 02/15 03/02 M L 06/09 06/02 06/13 02/13 07/05 06/04 06/03 04/10 05/00 00/02	GL Japanese) IBM-udcJP) user-definable characters.
01/11 02/04 02/08 04/03	GL KSC5601-1987.
01/11 02/04 02/09 03/00	GL CNS11643-1986-1.
01/11 02/04 02/10 03/01	GL CNS11643-1986-2.
01/11 02/05 02/15 03/00 M L 05/05 05/04 04/06 02/13 03/07 00/02	UCS-2 encoded as base64; used only for those characters not encoded by any of the other 7-bit escape sequences listed above.

When converting from a code set to fold7, the escape sequence used to designate the code set is chosen according to the order listed. For example, the JISX0208.1983-0 characters use **01/11 01/04 04/02** as the designation.

Files: The following list describes the fold7 converters that are found in the **/usr/lib/nls/loc/iconv** directory:

Converter	Description
fold7_IBM-850	Interchange format to IBM-850
fold7_IBM-921	Interchange format to IBM-921
fold7_IBM-922	Interchange format to IBM-922
fold7_IBM-932	Interchange format to IBM-932
fold7_IBM-943	Interchange format to IBM-943
fold7_IBM_1124	Interchange format to IBM-1124
fold7_IBM_1129	Interchange format to IBM-1129
fold7_IBM_eucCN	Interchange format to IBM-eucCN
fold7_IBM-eucJP	Interchange format to IBM-eucJP
fold7_IBM-eucKR	Interchange format to IBM-eucKR
fold7_IBM-eucTW	Interchange format to IBM-eucTW
fold7_ISO8859-1	Interchange format to ISO8859-1
fold7_ISO8859-2	Interchange format to ISO8859-2
fold7_ISO8859-3	Interchange format to ISO8859-3
fold7_ISO8859-4	Interchange format to ISO8859-4
fold7_ISO8859-5	Interchange format to ISO8859-5
fold7_ISO8859-6	Interchange format to ISO8859-6
fold7_ISO8859-7	Interchange format to ISO8859-7
fold7_ISO8859-8	Interchange format to ISO8859-8
fold7_ISO8859-9	Interchange format to ISO8859-9

Converter	Description
fold7_TIS-620	Interchange format to TIS-620
fold7_UTF-8	Interchange format to UTF-8
fold7_big5	Interchange format to big5
fold7_GBK	Interchange format to GBK
IBM-921_fold7	IBM-921 to interchange format
IBM-922_fold7	IBM-922 to interchange format
IBM-850_fold7	IBM-850 to interchange format
IBM-932_fold7	IBM-932 to interchange format
IBM-943_fold7	IBM-943 to interchange format
IBM-1124_fold7	IBM-1124 to interchange format
IBM-1129_fold7	IBM-1129 to interchange format
IBM-eucCN_fold7	IBM-eucCN to interchange format
IBM-eucJP_fold7	IBM-eucJP to interchange format
IBM-eucKR_fold7	IBM-eucKR to interchange format
IBM-eucTW_fold7	IBM-eucTW to interchange format
ISO8859-1_fold7	ISO8859-1 to interchange format
ISO8859-2_fold7	ISO8859-2 to interchange format
ISO8859-3_fold7	ISO8859-3 to interchange format
ISO8859-4_fold7	ISO8859-4 to interchange format
ISO8859-5_fold7	ISO8859-5 to interchange format
ISO8859-6_fold7	ISO8859-6 to interchange format
ISO8859-7_fold7	ISO8859-7 to interchange format
ISO8859-8_fold7	ISO8859-8 to interchange format
ISO8859-9_fold7	ISO8859-9 to interchange format
TIS-620_fold7	TIS-620 to interchange format
UTF-8_fold7	UTF-8 to interchange format
big5_fold7	big5 to interchange format
GBK_fold7	GBK to interchange format

List of Interchange Converters—8-bit

This converter provides conversions between internal code and 8-bit standard interchange formats (fold8). The fold8 name identifies encodings that can be used to pass text data through 8-bit mail protocols. The encodings are based on ISO2022. For more information about fold8 see “Understanding libiconv” on page 412.

The fold8 converters convert characters from a specific code set encoding to a canonical 8-bit encoding that identifies each character. This type of conversion is useful in networks where clients communicate with different code sets but use the same character sets. For example:

IBM-850 <—> **ISO8859-1**	Common Latin characters
IBM-932 <—>**IBM-eucJP**	Common Japanese characters

The following escape sequences designate standard code sets.

Escape Sequence	Standard Code Set
01/11 02/04 02/09 04/01	GR right half of GB2312.1980-0.
01/11 02/13 04/01	GR right half of ISO8859-1.
01/11 02/13 04/02	GR right half of ISO8859-2.
01/11 02/13 04/03	GR right half of ISO8859-3.
01/11 02/13 04/04	GR right half of ISO8859-4.
01/11 02/13 04/06	GR right half of ISO8859-7.
01/11 02/13 04/07	GR right half of ISO8859-6.
01/11 02/13 04/08	GR right half of ISO8859-8.
01/11 02/13 04/13	GR right half of ISO8859-5.
01/11 02/13 04/13	GR right half of ISO8859-9.
01/11 02/09 04/09	GR right half of JIS X0201.1976-1.
01/11 02/04 02/09 04/02	GR JIS X0208.1983-1.
01/11 02/04 02/09 04/00	GR JISX0208.1978-1.
01/11 02/09 04/02	GR 7-bit ASCII or left half of ISO8859-1.
01/11 02/05 02/15 03/01 M L 04/09 04/02 04/13 02/13 03/08 03/05 03/00 00/02	GR right half of IBM-850 unique characters. Characters common to ISO8859-1 should not use this escape sequence.
01/11 02/05 02/15 03/02 M L 04/09 04/02 04/13 02/13 07/05 06/04 06/03 04/10 05/00 00/02	GR right half of Japanese user-definable characters.
01/11 02/08 04/02	GL 7-bit ASCII or left half of ISO8859-1.
01/11 02/14 04/01	GL right half of ISO8859-1.
01/11 02/14 04/02	GL right half of ISO8859-2.
01/11 02/14 04/03	GL right half of ISO8859-3.
01/11 02/14 04/04	GL right half of ISO8859-4.
01/11 02/14 04/06	GL right half of ISO8859-7.
01/11 02/14 04/07	GL right half of ISO8859-6.
01/11 02/14 04/08	GL right half of ISO8859-8.
01/11 02/14 04/12	GL right half of ISO8859-5.
01/11 02/14 04/13	GL right half of ISO8859-9.
01/11 02/08 04/09	GL right half of JIS X0201.1976-0.
01/11 02/08 04/10	GL left half of JIS X0201.1976.
01/11 02/04 02/08 04/02	GL JIS X0208.1983-0.
01/11 02/04 04/02	GL JIS X0208.1983-0.
01/11 02/04 04/00	GL JIS X0208.1978-0.
01/11 02/05 02/15 03/01 M L 06/09 06/02 06/13 02/13 03/08 03/05 03/00 00/02	GL right half of IBM-850 unique characters. Characters common to ISO8859-1 do not use this escape sequence.
01/11 02/05 02/15 03/02 M L 06/09 06/02 06/13 02/13 07/05 06/04 06/03 04/10 05/00 00/02	GL Japanese (IBM-udcJP) user-definable characters.
01/11 02/04 02/09 04/03	GR KSC5601-1987.
01/11 02/04 02/09 03/00	GR CNS11643-1986-1.
01/11 02/04 02/10 03/01	GR CNS11643-1986-2.

Escape Sequence	Standard Code Set
01/11 02/05 02/15 03/02 M L 04/09 04/02 04/13 02/13 07/05 06/04 06/03 05/05 05/08 00/02	GR right half of Traditional Chinese user-definable characters.
01/11 02/05 02/15 03/02 M L 04/09 04/02 04/13 02/13 07/03 06/02 06/04 05/05 05/08 00/02	GR right half of IBM-850 unique symbols.
01/11 02/04 02/08 04/03	GL KSC5601-1987.
01/11 02/05 02/15 03/02 M L 06/09 06/02 06/13 02/13 07/05 06/04 06/03 05/05 05/08 00/02	GL Traditional Chinese (IBM-udcTW) user-definable characters.
01/11 02/05 02/15 03/02 M L 06/09 06/02 06/13 02/13 07/03 06/02 06/04 05/05 05/08 00/02	GL Traditional Chinese IBM-850 unique symbols (IBM-shdTW) user-definable characters.
01/11 02/05 02/15 03/00 M L 05/05 05/04 04/06 02/13 03/08 00/02	UCS-2 encoded as UTF-8; used only for those characters not encoded by any of the above escape sequences listed above.

When converting from a code set to fold8, the escape sequence used to designate the code set is chosen according to the order listed. For example, the JISX0208.1983-0 characters use **01/11 02/04 02/08 04/02** as the designation.

Files: The following list describes the fold8 converters found in the **/usr/lib/nls/loc/iconv** directory:

Converter	Description
fold8_IBM-850	Interchange format to IBM-850
fold8_IBM-921	Interchange format to IBM-921
fold8_IBM-922	Interchange format to IBM-922
fold8_IBM-932	Interchange format to IBM-932
fold8_IBM-943	Interchange format to IBM-943
fold8_IBM-1124	Interchange format to IBM-1124
fold8_IBM-1129	Interchange format to IBM-1129
fold8_IBM-eucCN	Interchange format to IBM-eucCN
fold8_IBM-eucJP	Interchange format to IBM-eucJP
fold8_IBM-eucKR	Interchange format to IBM-eucKR
fold8_IBM-eucTW	Interchange format to IBM-eucTW
fold8_IBM-eucCN	Interchange fromat to IBM-eucCN
fold8_ISO8859-1	Interchange format to ISO8859-1
fold8_ISO8859-2	Interchange format to ISO8859-2
fold8_ISO8859-3	Interchange format to ISO8859-3
fold8_ISO8859-4	Interchange format to ISO8859-4
fold8_ISO8859-5	Interchange format to ISO8859-5
fold8_ISO8859-6	Interchange format to ISO8859-6
fold8_ISO8859-7	Interchange format to ISO8859-7
fold8_ISO8859-8	Interchange format to ISO8859-8
fold8_ISO8859-9	Interchange format to ISO8859-9
fold8_TIS-620	Interchange format to TIS-620
fold8_UTF-8	Interchange format to UTF-8
fold8_big5	Interchange format to big5

Converter	Description
fold8_GBK	Interchange format to GBK
IBM-921_fold8	IBM-921 to interchange format
IBM-922_fold8	IBM-922 to interchange format
IBM-850_fold8	IBM-850 to interchange format
IBM-932_fold8	IBM-932 to interchange format
IBM-943_fold8	IBM-943 to interchange format
IBM-1124_fold8	IBM-1124 to interchange format
IBM-1129_fold8	IBM-1129 to interchange format
IBM-eucCN_fold8	IBM-eucCN to interchange format
IBM-eucJP_fold8	IBM-eucJP to interchange format
IBM-eucKR_fold8	IBM-eucKR to interchange format
IBM-eucTW_fold8	IBM-eucTW to interchange format
IBM-eucCN_fold8	IBM-eucCN to interchange format
ISO8859-1_fold8	ISO8859-1 to interchange format
ISO8859-2_fold8	ISO8859-2 to interchange format
ISO8859-3_fold8	ISO8859-3 to interchange format
ISO8859-4_fold8	ISO8859-4 to interchange format
ISO8859-5_fold8	ISO8859-5 to interchange format
ISO8859-6_fold8	ISO8859-6 to interchange format
ISO8859-7_fold8	ISO8859-7 to interchange format
ISO8859-8_fold8	ISO8859-8 to interchange format
ISO8859-9_fold8	ISO8859-9 to interchange format
TIS-620_fold8	TIS-620 to interchange format
UTF-8_fold8	UTF-8 to interchange format
big5_fold8	big5 to interchange format
GBK_fold8	GBK to interchange format

List of Interchange Converters—Compound Text

Compound text interchange converters convert between compound text and internal code sets.

Compound text is an interchange encoding defined by the X Consortium. It is used to communicate text between X clients. Compound text is based on ISO2022 and can encode most character sets using standard escape sequences. It also provides extensions for encoding private character sets. The supported code sets provide a converter to and from compound text. The name used to identify the compound text encoding is ct.

The following escape sequences are used to designate standard code sets in the order listed below.

01/11 02/05 02/15 03/01 M L 04/09 04/02 04/13 02/13 03/08 03/05 03/00 00/02
GR right half of IBM-850 unique characters. Characters common to ISO8859-1 should not use this escape sequence.

01/11 02/05 02/15 03/02 M L 04/09 04/02 04/13 02/13 07/05 06/04 06/03 04/10 05/00 00/02
GR right half of Japanese user-definable characters.

01/11 02/05 02/15 03/01 M L 06/09 06/02 06/13 02/13 03/08 03/05 03/00 00/02
GL right half of IBM-850 unique characters. Characters common to ISO8859-1 do not use this escape sequence.

01/11 02/05 02/15 03/02 M L 06/09 06/02 06/13 02/13 07/05 06/04 06/03 04/10 05/00 00/02
GL Japanese (IBM-udcJP) user-definable characters.

Files: The following list describes the compound text converters that are found in the **/usr/lib/nls/loc/iconv** directory:

Converter	Description
ct_IBM-850	Interchange format to IBM-850
ct_IBM-921	Interchange format to IBM-921
ct_IBM-922	Interchange format to IBM-922
ct_IBM-932	Interchange format to IBM-932
ct_IBM-943	Interchange format to IBM-943
ct_IBM-1124	Interchange format to IBM-1124
ct_IBM-1129	Interchange format to IBM-1129
ct_IBM-eucCN	Interchange format to IBM-eucCN
ct_IBM-eucJP	Interchange format to IBM-eucJP
ct_IBM-eucKR	Interchange format to IBM-eucKR
ct_IBM-eucTW	Interchange format to IBM-eucTW
ct_ISO8859-1	Interchange format to ISO8859-1
ct_ISO8859-2	Interchange format to ISO8859-2
ct_ISO8859-3	Interchange format to ISO8859-3
ct_ISO8859-4	Interchange format to ISO8859-4
ct_ISO8859-5	Interchange format to ISO8859-5
ct_ISO8859-6	Interchange format to ISO8859-6
ct_ISO8859-7	Interchange format to ISO8859-7
ct_ISO8859-8	Interchange format to ISO8859-8
ct_ISO8859-9	Interchange format to ISO8859-9
ct_TIS-620	Interchange format to TIS-620
ct_big5	Interchange format to big5
ct_GBK	Interchange format to GBK
IBM-850_ct	IBM-850 to interchange format
IBM-921_ct	IBM-921 to interchange format
IBM-922_ct	IBM-922 to interchange format
IBM-932_ct	IBM-932 to interchange format
IBM-943_ct	IBM-943 to interchange format
IBM-1124_ct	IBM-1124 to interchange format
IBM-1129_ct	IBM-1129 to interchange format
IBM-eucCN_ct	IBM-eucCN to interchange format
IBM-eucJP_ct	IBM-eucJP to interchange format
IBM-eucKR_ct	IBM-eucKR to interchange format
IBM-eucTW_ct	IBM-eucTW to interchange format

Converter	Description
ISO8859-1_ct	ISO8859-1 to interchange format
ISO8859-2_ct	ISO8859-2 to interchange format
ISO8859-3_ct	ISO8859-3 to interchange format
ISO8859-4_ct	ISO8859-4 to interchange format
ISO8859-5_ct	ISO8859-5 to interchange format
ISO8859-6_ct	ISO8859-6 to interchange format
ISO8859-7_ct	ISO8859-7 to interchange format
ISO8859-8_ct	ISO8859-8 to interchange format
ISO8859-9_ct	ISO8859-9 to interchange format
TIS-620_ct	TIS-620 to interchange format
big5_ct	big5 to interchange format
GBK_ct	GBK to interchange format

List of Interchange Converters—uucode

This converter provides the same mapping as the **uuencode** and **uudecode** Command.

During conversion from uucode, 62 bytes at a time (including a new-line character trailing the record) are converted, and generating 45 bytes in *outbuf*.

Files: The following list describes the uucode converters found in the **/usr/lib/nls/loc/iconv** directory:

Converter	Description
IBM-850_uucode	IBM-850 to uucode
IBM-921_uucode	IBM-921 to uucode
IBM-922_uucode	IBM-922 to uucode
IBM-932_uucode	IBM-932 to uucode
IBM-943_uucode	IBM-943 to uucode
IBM-1124_uucode	IBM-1124 to uucode
IBM-1129_uucode	IBM-1129 to uucode
IBM-eucJP_uucode	IBM-eucJP to uucode
IBM-eucKR_uucode	IBM-eucKR to uucode
IBM-eucTW_uucode	IBM-eucTW to uucode
IBM-eucCN_uucode	IBM-eucCN to uucode
ISO8859-1_uucode	ISO8859-1 to uucode
ISO8859-2_uucode	ISO8859-2 to uucode
ISO8859-3_uucode	ISO8859-3 to uucode
ISO8859-4_uucode	ISO8859-4 to uucode
ISO8859-5_uucode	ISO8859-5 to uucode
ISO8859-6_uucode	ISO8859-6 to uucode
ISO8859-7_uucode	ISO8859-7 to uucode
ISO8859-8_uucode	ISO8859-8 to uucode
ISO8859-9_uucode	ISO8859-9 to uucode

Converter	Description
TIS-620_uucode	TIS-620 to uucode
big5_uucode	big5 to uucode
GBK_uucode	GBK to uucode
uucode_IBM-850	uucode to IBM-850
uucode_IBM-921	uucode to IBM-921
uucode_IBM-922	uucode to IBM-922
uucode_IBM-932	uucode to IBM-932
uucode_IBM-943	uucode to IBM-943
uucode_IBM-1124	uucode to IBM-1124
uucode_IBM-1129	uucode to IBM-1129
uucode_IBM-eucCN	uucode to IBM-eucCN
uucode_IBM-eucJP	uucode to IBM-eucJP
uucode_IBM-eucKR	uucode to IBM-eucKR
uucode_IBM-eucTW	uucode to IBM-eucTW
uucode_ISO8859-1	uucode to ISO8859-1
uucode_ISO8859-2	uucode to ISO8859-2
uucode_ISO8859-3	uucode to ISO8859-3
uucode_ISO8859-4	uucode to ISO8859-4
uucode_ISO8859-5	uucode to ISO8859-5
uucode_ISO8859-6	uucode to ISO8859-6
uucode_ISO8859-7	uucode to ISO8859-7
uucode_ISO8859-8	uucode to ISO8859-8
uucode_ISO8859-9	uucode to ISO8859-9
uucode_TIS-1124	uucode to TIS-1129
uucode_big5	uucode to big5
uucode_GBK	uucode to GBK

List of UCS-2 Interchange Converters

UCS-2 is a universal, 16-bit encoding described in the “Code Set Overview” on page 379. Conversions for each code set are provided in both directions, between the code set and UCS-2.

UCS-2 converters are found in **/usr/lib/nls/loc/uconvTable** and **/usr/lib/nls/loc/uconv** directories. The **uconvdef** command is used to generate new converters or to customize existing UCS-2 converters.

The /usr/lib/nls/loc/iconv/Universal_UCS_Conv converter is used to generate conversions from any code set `X` to code set `Y` by setting the proper links:

```
cd /usr/lib/nls/loc/iconv
ln -s /usr/lib/nls/loc/uconv/Universal_UCS_Conv X_Y
ln -s /usr/lib/nls/loc/uconv/UCSTBL X_UCS-2
ln-s /usr/lib/nls/loc/uconv/UCSTBL UCS-2_Y
ln -s /usr/lib/nls/loc/uconv/UCSTBL X
ln -s /usr/lib/nls/loc/uconv/UCSTBL Y
```

Converter	Description
ISO8859-1	UCS-2 <—> ISO Latin-1

Converter	Description
ISO8859-2	UCS-2 <—> ISO Latin-2
ISO8859-3	UCS-2 <—> ISO Latin-3
ISO8859-4	UCS-2 <—> ISO Latin-4
ISO8859-5	UCS-2 <—> ISO Cyrillic
ISO8859-6	UCS-2 <—> ISO Arabic
ISO8859-7	UCS-2 <—> ISO Greek
ISO8859-8	UCS-2 <—> ISO Hebrew
ISO8859-9	UCS-2 <—> ISO Turkish
JISX0201.1976-0	UCS-2 <—> Japanese JISX0201-0
JISX0208.1983-0	UCS-2 <—> Japanese JISX0208-0
CNS11643.1986-1	UCS-2 <—> Chinese CNS11643-1
CNS11643.1986-2	UCS-2 <—> Chinese CNS11643-2
KSC5601.1987-0	UCS-2 <—> Korean KSC5601-0
IBM-eucCN	UCS-2 <—> Simplified Chinese EUC
IBM-udcCN	UCS-2 <—> Simplified Chinese user-defined characters
IBM-sbdCN	UCS-2 <—> Simplified Chinese IBM-specific characters
GB2312.1980-0	UCS-2 <—> Simplified Chinese GB
IBM-1381	UCS-2 <—> Simplified Chinese PC data code
IBM-935	UCS-2 <—> Simplified Chinese EBCDIC
IBM-936	UCS-2 <—> Simplified Chinese PC5550
IBM-eucJP	UCS-2 <—> Japanese EUC
IBM-eucKR	UCS-2 <—> Korean EUC
IBM-eucTW	UCS-2 <—> Traditional Chinese EUC
IBM-udcJP	UCS-2 <—> Japanese user-defined characters
IBM-udcTW	UCS-2 <—> Traditional Chinese user-defined characters
IBM-sbdTW	UCS-2 <—> Traditional Chinese IBM-specific characters
UTF-8	UCS-2 <—> UTF-8
IBM-437	UCS-2 <—> USA PC data code
IBM-850	UCS-2 <—> Latin-1 PC data code
IBM-852	UCS-2 <—> Latin-2 PC data code
IBM-857	UCS-2 <—> Turkish PC data code
IBM-860	UCS-2 <—> Portuguese PC data code
IBM-861	UCS-2 <—> Icelandic PC data code
IBM-863	UCS-2 <—> French Canadian PC data code
IBM-865	UCS-2 <—> Nordic PC data code
IBM-869	UCS-2 <—> Greek PC data code
IBM-921	UCS-2 <—> Baltic Multilingual data code
IBM-922	UCS-2 <—> Estonian data code
IBM-932	UCS-2 <—> Japanese PC data code
IBM-943	UCS-2 <—> Japanese PC data code
IBM-934	UCS-2 <—> Korea PC data code

Converter	Description
IBM-936	UCS-2 <—> People's Republic of China PC data code
IBM-938	UCS-2 <—> Taiwanese PC data code
IBM-942	UCS-2 <—> Extended Japanese PC data code
IBM-944	UCS-2 <—> Korean PC data code
IBM-946	UCS-2 <—> People's Republic of China SAA data code
IBM-948	UCS-2 <—> Traditional Chinese PC data code
IBM-1124	UCS-2 <—> Ukranian PC data code
IBM-1129	UCS-2 <—> Vietnamese PC data code
TIS-620	UCS-2 <—> Thailand PC data code
IBM-037	UCS-2 <—> USA, Canada EBCDIC
IBM-273	UCS-2 <—> Germany, Austria EBCDIC
IBM-277	UCS-2 <—> Denmark, Norway EBCDIC
IBM-278	UCS-2 <—> Finland, Sweden EBCDIC
IBM-280	UCS-2 <—> Italy EBCDIC
IBM-284	UCS-2 <—> Spain, Latin America EBCDIC
IBM-285	UCS-2 <—> United Kingdom EBCDIC
IBM-297	UCS-2 <—> France EBCDIC
IBM-500	UCS-2 <—> International EBCDIC
IBM-875	UCS-2 <—> Greek EBCDIC
IBM-930	UCS-2 <—> Japanese Katakana-Kanji EBCDIC
IBM-933	UCS-2 <—> Korean EBCDIC
IBM-937	UCS-2 <—> Traditional Chinese EBCDIC
IBM-939	UCS-2 <—> Japanese Latin-Kanji EBCDIC
IBM-1026	UCS-2 <—> Turkish EBCDIC
IBM-1112	UCS-2 <—> Baltic Multilingual EBCDIC
IBM-1122	UCS-2 <—> Estonian EBCDIC
IBM-1124	UCS-2 <—> Ukranian EBCDIC
IBM-1129	UCS-2 <—> Vietnamese EBCDIC
GBK	UCS-2<—> Simplified Chinese
TIS-620	UCS-2 <—>Thailand EBCDIC

List of UTF-8 Interchange Converters

UTF-8 is a universal, multibyte encoding described in the "UCS-2 and UTF-8" on page 407. Conversions for each code set are provided in both directions, between the code set and UTF-8.

UTF-8 converters are usually done by using the Universal_UCS_Conv (see "List of UCS-2 Interchange Converters" on page 433 and **/usr/lib/nls/loc/uconv/UTF-8** conversion.

Converter	Description
ISO8859-1	UTF-8 <—> ISO Latin-1
ISO8859-2	UTF-8 <—> ISO Latin-2
ISO8859-3	UTF-8 <—> ISO Latin-3
ISO8859-4	UTF-8 <—> ISO Latin-4

Converter	Description
ISO8859-5	UTF-8 <—> ISO Cyrillic
ISO8859-6	UTF-8 <—> ISO Arabic
ISO8859-7	UTF-8 <—> ISO Greek
ISO8859-8	UTF-8 <—> ISO Hebrew
ISO8859-9	UTF-8 <—> ISO Turkish
JISX0201.1976-0	UTF-8 <—> Japanese JISX0201-0
JISX0208.1983-0	UTF-8 <—> Japanese JISX0208-0
CNS11643.1986-1	UTF-8 <—> Chinese CNS11643-1
CNS11643.1986-2	UTF-8 <—> Chinese CNS11643-2
KSC5601.1987-0	UTF-8 <—> Korean KSC5601-0
IBM-eucCN	UTF-8 <—> Simplified Chinese EUC
IBM-eucJP	UTF-8 <—> Japanese EUC
IBM-eucKR	UTF-8 <—> Korean EUC
IBM-eucTW	UTF-8 <—> Traditional Chinese EUC
IBM-udcJP	UTF-8 <—> Japanese user-defined characters
IBM-udcTW	UTF-8 <—> Traditional Chinese user-defined characters
IBM-sbdTW	UTF-8 <—> Traditional Chinese IBM-specific characters
UCS-2	UTF-8 <—> UCS-2
IBM-437	UTF-8 <—> USA PC data code
IBM-850	UTF-8 <—> Latin-1 PC data code
IBM-852	UTF-8 <—> Latin-2 PC data code
IBM-857	UTF-8 <—> Turkish PC data code
IBM-860	UTF-8 <—> Portuguese PC data code
IBM-861	UTF-8 <—> Icelandic PC data code
IBM-863	UTF-8 <—> French Canadian PC data code
IBM-865	UTF-8 <—> Nordic PC data code
IBM-869	UTF-8 <—> Greek PC data code
IBM-921	UTF-8 <—> Baltic Multilingual data code
IBM-922	UTF-8 <—> Estonian data code
IBM-932	UTF-8 <—> Japanese PC data code
IBM-943	UTF-8 <—> Japanese PC data code
IBM-934	UTF-8 <—> Korea PC data code
IBM-935	UTF-8 <—> Simplified Chinese EBCDIC
IBM-936	UTF-8 <—> People’s Republic of China PC data code
IBM-938	UTF-8 <—> Taiwanese PC data code
IBM-942	UTF-8 <—> Extended Japanese PC data code
IBM-944	UTF-8 <—> Korean PC data code
IBM-946	UTF-8 <—> People’s Republic of China SAA data code
IBM-948	UTF-8 <—> Traditional Chinese PC data code
IBM-1124	UTF-8 <—> Ukranian PC data code
IBM-1129	UTF-8 <—> Vietnamese PC data code

Converter	Description
TIS-620	UTF-8 <—> Thailand PC data code
IBM-037	UTF-8 <—> USA, Canada EBCDIC
IBM-273	UTF-8 <—> Germany, Austria EBCDIC
IBM-277	UTF-8 <—> Denmark, Norway EBCDIC
IBM-278	UTF-8 <—> Finland, Sweden EBCDIC
IBM-280	UTF-8 <—> Italy EBCDIC
IBM-284	UTF-8 <—> Spain, Latin America EBCDIC
IBM-285	UTF-8 <—> United Kingdom EBCDIC
IBM-297	UTF-8 <—> France EBCDIC
IBM-500	UTF-8 <—> International EBCDIC
IBM-875	UTF-8 <—> Greek EBCDIC
IBM-930	UTF-8 <—> Japanese Katakana-Kanji EBCDIC
IBM-933	UTF-8 <—> Korean EBCDIC
IBM-937	UTF-8 <—> Traditional Chinese EBCDIC
IBM-939	UTF-8 <—> Japanese Latin-Kanji EBCDIC
IBM-1026	UTF-8 <—> Turkish EBCDIC
IBM-1112	UTF-8 <—> Baltic Multilingual EBCDIC
IBM-1122	UTF-8 <—> Estonian EBCDIC
IBM-1124	UTF-8 <—> Ukranian EBCDIC
IBM-1129	UTF-8 <—> Vietnamese EBCDIC
IBM-1381	UTF-8 <—> Simplified Chinese PC data code
GBK	UTF-8<—> Simplified Chinese
TIS-620	UTF-8 <—> Thailand EBCDIC

List of Miscellaneous Converters

A set of low level converters used by the code set and interchange converters is provided. These converters are called miscellaneous converters. These low-level converters may be used by some of the interchange converters. However, the use of these converters is discouraged because they are intended for support of other converters.

Files: The following list describes the miscellaneous converters found in the **/usr/lib/nls/loc/iconv** and **/usr/lib/nls/loc/iconvTable** directories:

Converter	Description
IBM-932_JISX0201.1976-0	IBM-932 to JISX0201.1976-0
IBM-932_JISX0208.1983-0	IBM-932 to JISX0208.1983-0
IBM-932_IBM-udcJP	IBM-932 to IBM-udcJP (Japanese user-defined characters)
IBM-943_JISX0201.1976-0	IBM-943 to JISX0201.1976-0
IBM-943_JISX0208.1983-0	IBM-943 to JISX0208.1983-0
IBM-943_IBM-udcJP	IBM-943 to IBM-udcJP (Japanese user-defined characters
IBM-eucJP_JISX0201.1976-0	IBM-eucJP to JISX0201.1976-0
IBM-eucJP_JISX0208.1983-0	IBM-eucJP to JISX0208.1983-0
IBM-eucJP_IBM-udcJP	IBM-eucJP to IBM-udcJP (Japanese user-defined characters)

Converter	Description
IBM-eucKR_KSC5601.1987-0	IBM_eucKR to KSC5601.1987-0
IBM-eucTW_CNS11643.1986-1	IBM-eucTW to CNS11643.1986.1
IBM-eucTW_CNS11643.1986-2	IBM-eucTW to CNS11643.1986-2
IBM-eucCN_GB2312.1980-0	IBM-eucCN to GB2312.1980-0

Writing Converters Using the iconv Interface

This section provides a general background on the **iconv** subroutines and structures in preparation for writing code set converters. This section gives an overview of the control flow and the order in which the framework operates, details about writing code set converters, and an example including the code, header file, and a makefile. This section applies to the **iconv** framework within AIX.

Under the framework of the **iconv_open**, **iconv** and **iconv_close** subroutines, you can create and use several different types of converters. Applications can call these subroutines to convert characters in one code set into characters in a different code set. The access and use of the **iconv_open**, **iconv** and **iconv_close** subroutines is standardized by X/Open XPG4.

Code Sets and Converters

Code sets can be classified into two categories: stateful encodings and stateless encodings.

Stateful Code Sets and Converters

The stateful encodings use shift-in and shift-out codes to change state. For instance, the shift-out can be used to indicate the start of host double-byte data in a data stream of characters, and shift-in can be used to indicate the end of this double-byte character data. When the double-byte data is off, it signals the start of single-byte character data. An example of such a stateful code set is IBM-930 used mainly on mainframes (hosts).

Converters written to do the conversion of stateful encodings to other code sets tend to be complex due to the extra processing needed.

Stateless Code Sets and Converters

The stateless code sets are those that can be classified as one of two types:

- Single-byte code sets, such as ISO8859 family (ISO8859-1, ISO8859-2, and so on)
- Multibyte code sets, such as IBM-eucJP (Japanese), IBM-932 (Shift-JIS).

Note that conversions are meaningful only if the code sets represent the same characters.

The simplest types of code set conversion can be found in single-byte code set converters, such as the converter from ISO8859-1 to IBM-850. These single-byte code set converters are based on simple table-based conversions. The conversion of multibyte character encodings, such as IBM-eucJP to IBM-932, are in general based on an algorithm and not on tables, because the tables can get lengthy.

iconv Framework - Overview of Structures

The **iconv** framework consists of the **iconv_open**, **iconv** and **iconv_close** subroutines. It is based on a common core structure that is part of all converters. The core structure is initialized at the load time of the converter object module. After the loading of the converter is complete, the main entry point, which is always the **instantiate** subroutine, is invoked. This initializes the core structure and returns the core converter descriptor. This is further used during the call to the **init** subroutine provided by the converter to allocate the converter-specific structures. This **init** subroutine returns another converter descriptor that has a pointer to the core converter descriptor. The **init** subroutine allocates memory as needed and may

invoke other converters if needed. The **init** subroutine is the place for any converter-specific initialization whereas the **instantiate** subroutine is a generic entry point.

Once the converter descriptor for this converter is allocated and initialized, the next step is to provide the actual code needed for the **exec** part of the functionality. If the converter is a table-based converter, the only need is to provide a source file format that conforms to the input needs of the **genxlt** utility, which takes this source table as the input and generates an output file format usable by the **iconv** framework.

iconv.h File and Structures

The **iconv.h** file in **/usr/include** defines the following structures:

```
typedef struct __iconv_rec   iconv_rec, *iconv_t;
struct __iconv_rec   {
    _LC_object_t  hdr;
    iconv_t (*open)(const char *tocode, const char *fromcode);
    size_t (*exec)(iconv_t cd, char **inbuf, size_t *inbytesleft,
            char **outbuf, size_t *outbytesleft);
    void  (*close)(iconv_t cd);
};
```

The common core structure is as follows (**/usr/include/iconv.h**):

```
typedef struct _LC_core_iconv_type      _LC_core_iconv_t;
struct _LC_core_iconv_type {
        _LC_object_t  hdr;
        /* implementation initialization */
        _LC_core_iconv_t        *(*init)();
        size_t  (*exec)();
        void    (*close)();
};
```

Every converter has a static memory area which contains the **_LC_core_iconv_t** structure. It is initialized in the **instantiate** subroutine provided as part of the converter program.

iconv Control Flow

The following sections describe the **iconv** control flow.

The iconv_open Subroutine: An application invokes a code set converter by the following call:

```
iconv_open(char *to_codeset, char *from_codeset)
```

The *to* and *from* code sets are used in selecting the converter by way of the search path defined by the **LOCPATH** environment variable. The **iconv_open** subroutine uses the **_lc_load** subroutine to load the object module specified by concatenating the *from* and *to* code set names to the **iconv_open** subroutine.

```
CONVERTER NAME= "from_codeset" + "_" +"to_codeset"
```

If the *from_codeset* is IBM-850 and the *to_codeset* is ISO8859-1, the converter name is IBM-850_ISO8859-1.

After loading the converter, its entry point is invoked by the **_lc_load** loader subroutine. This is the first call to the converter. The **instantiate** subroutine then initializes the **_LC_core_iconv_t** core structure. The **iconv_open** subroutine then calls the **init** subroutine associated with the core structure thus returned. The **init** subroutine is responsible for allocating the converter specific descriptor structure and initializing it as needed by the converter. The **iconv_open** subroutine returns this converter-specific structure. However, the return value is typecast to **iconv_t** in the user's application. Thus, the application does not see the whole of the converter-specific structure; it sees only the public **iconv_t** structure. The converter code itself uses the private converter structure. Applications that use **iconv** converters should not change the converter descriptor; the converter descriptor should be used as an opaque structure.

Entry Point: An entry point is declared in every converter such that when the converter is opened by a call to the **iconv_open** subroutine, that entry point is automatically invoked. The entry point is the **instantiate** subroutine that should be provided in all converters. The entry point is specified in the makefile as follows:

```
LDENTRY=-einstantiate
```

When the converter is loaded on a call to **iconv_open()**, the instantiate subroutine is invoked. This subroutine knows how the converter works. It initializes a static core conversion descriptor structure **_LC_core_iconv_t cd**.

The core conversion descriptor **cd** contains pointers to the **init**, **_iconv_exec**, and **_iconv_close** subroutines supplied by the specific converter. The **instantiate** subroutine returns the core conversion descriptor to be used later. The **_LC_core_iconv_t** structure is defined in **/usr/include/iconv.h**.

When the **iconv_open** subroutine is called, the following actions occur:

1. The converter is found using **LOCPATH**, the converter is loaded, and the **instantiate** subroutine is invoked. On success, it returns the core conversion descriptor. (**_LC_core_iconv_t *cd**). The **instantiate** subroutine provided by the converter is responsible for initializing the header in the core structure.
2. The **iconv_open** subroutine then invokes the **init** subroutine specified in the core conversion descriptor. The **init** subroutine provided by the converter is responsible for allocation of memory needed to hold the converter descriptor needed for this specific converter. For example, the following may be the structure needed by a stateless converter:

```
typedef struct _LC_sample_iconv_rec {
                  LC_core_iconv_t        core;

            } _LC_sample_iconv_t;

To initialize this, the converter has to do the following in the
init subroutine:

static _LC_sample_iconv_t*
init (_LC_core_iconv_t *core_cd, char* toname,char* fromname)
{
   _LC_sample_iconv_t     *cd;    /* converter descriptor */

    /*
    **        Allocate a converter descriptor
    **/
    if(!(cd = ( _LC_sample_iconv_t *) malloc (
                    sizeof(_LC_sample_iconv_t ))))
            return (NULL);

    /*
    ** Copy the core part of converter descriptor which is
    ** passed in
    */
    cd->core = *core_cd;
    /*
    **      Return the converter descriptor
    */
    return cd;
}
```

The iconv Subroutine: An application invokes the **iconv** subroutine to do the actual code set conversions. The **iconv** subroutine invokes the **exec** subroutine in the core structure.

The iconv_close Subroutine: An application invokes the **iconv_close** subroutine to free any memory allocated for conversions. The **iconv_close** subroutine invokes the **close** subroutine in the core structure.

Writing a Code Set Converter

This section gives details on how to write a converter using the concepts explained so far. This is done starting with a simple converter and proceeds to a more complex one. The following procedures are discussed here:

- How to write an algorithmic converter.
- How to write a table lookup converter.
- How to write a stateful code set converter.

Every converter should define the following subroutines:

- **instantiate()**
- **init()**
- **iconv_exec()**
- **iconv_close()**

The converter-specific structure should have the core **iconv** structure as its first element. For example:

Example 1:

```
typedef struct _LC_example_rec {
    /* Core should be the first element */
        _LC_core_iconv_t        core;
    /* The rest are converter specific data (optional) */
        iconv_t         curcd;
        iconv_t         sb_cd;
        iconv_t         db_cd;
        unsigned char   *cntl;
} _LC_example_iconv_t;
```

Example 2: A simpler converter structure

```
typedef struct _LC_sample_iconv_rec {
        _LC_core_iconv_t        core;
} _LC_sample_iconv_t;
```

Stateless Converters - Algorithm Based

Every converter should have the subroutines previously specified. Only the subroutine headers are provided without details, except for the **instantiate** subroutine that is common to all converters and should be coded the same.

The following example of an algorithm-based stateless converter is a sample converter of the IBM-850 code set to the ISO8859-1 code set.

```
#include <stdlib.h>
#include <iconv.h>
#include "850_88591.h"
/*
 *      Name :  _iconv_exec()
 *
 *      This contains actual conversion method.
 */
static size_t   _iconv_exec(_LC_sample_iconv_t *cd,
                        unsigned char** inbuf,
                        size_t *inbytesleft,
                        unsigned char** outbuf,
                        size_t *outbytesleft)
/*
 *      cd              : converter descriptor
 *      inbuf           : input buffer
 *      outbuf          : output buffer
 *      inbytesleft     : number of data(in bytes) in input buffer
```

```
 *      outbytesleft     : number of data(in bytes) in output buffer
 */
{
}

/*
 *      Name :   _iconv_close()
 *
 *      Free the allocated converter descriptor
 */
static void      _iconv_close(iconv_t cd)
{
}

/*
 *      Name :  init()
 *
 *      This allocates and initializes the converter descriptor.
 */
static _LC_sample_iconv_t        *init (_LC_core_iconv_t *core_cd,
                                  char* toname, char* fromname)
{
}

/*
 *       Name :  instantiate()
 *
 *       Core part of a converter descriptor is initialized here.
 */
_LC_core_iconv_t         *instantiate(void)
{
        static _LC_core_iconv_t   cd;

        /*
        * * Initialize _LC_MAGIC and _LC_VERSION are
        ** defined in <lc_core.h>. _LC_ICONV and _LC_core_iconv_t
        ** are defined in <iconv.h>.
         */
        cd.hdr.magic = _LC_MAGIC;
        cd.hdr.version = _LC_VERSION;
        cd.hdr.type_id = _LC_ICONV;
        cd.hdr.size = sizeof (_LC_core_iconv_t);

        /*
         *      Set pointers to each method.
         */
        cd.init = init;
        cd.exec = _iconv_exec;
        cd.close = _iconv_close;

        /*
         *      Returns the core part
         */
        return &cd;
}
```

Stateful Converters

Here, only the subroutine headers are provided without details, except for the **instantiate** subroutine that is common to all converters and should be coded the same. Because stateful converters need more information, they provide additional converter-dependent information as well.

The following example of a stateful converter is a sample converter of IBM-930 to IBM-932 code set.

The `host.h` file contains the following structure:

```
typedef struct _LC_host_iconv_rec {
                _LC_core_iconv_t        core;
                iconv_t         curcd;
                iconv_t         sb_cd;
                iconv_t         db_cd;
                unsigned char   *cntl;
        } _LC_host_iconv_t;

#include <stdlib.h>
#include <sys/types.h>
#include <iconv.h>
#include "host.h"

/*
** The _iconv_exec subroutine to be invoked via cd->exec()
*/
static int      _iconv_exec(_LC_host_iconv_t *cd,
        unsigned char **inbuf, size_t *inbytesleft,
        unsigned char **outbuf, size_t *outbytesleft)
{
        unsigned char    *in, *out;
        int              ret_value;

    if (!cd){
        errno = EBADF; return NULL;
    }

    if (!inbuf) {
        cd->curcd = cd->sb_cd;
        return ICONV_DONE;
    }

    do {
    if ((ret_value = iconv(cd->curcd, inbuf, inbytesleft, outbuf,
             outbytesleft)) != ICONV_INVAL)
             return ret_value;
        in = *inbuf;
        out = *outbuf;
        if (in[0] == SO) {
            if (cd->curcd == cd->db_cd){
                errno = EILSEQ;
                return ICONV_INVAL;
            }
            cd->curcd = cd->db_cd;
        }
        else if (in[0] == SI) {
            if (cd->curcd == cd->sb_cd){
                errno = EILSEQ;
                return ICONV_INVAL;
            }
                cd->curcd = cd->sb_cd;
        }else if (in[0] <= 0x3f &&
            cd->curcd == cd->sb_cd) {
            if (*outbytesleft < 1){
                errno = E2BIG;
                return ICONV_OVER;
            }
                out[0] = cd->cntl[in[0]];
                *outbuf = ++out;
                (*outbytesleft)--;
        }
        else {
            errno = EILSEQ; return ICONV_INVAL;
        }
        *inbuf = ++in;
        (*inbytesleft)--;
    } while (1);
}

/*
```

```
** The iconv_close subroutine is a macro accessing this
** subroutine as set in the core iconv structure.
*/
static void   _iconv_close(_LC_host_iconv_t *cd)
{
    if (cd) {
        if (cd->sb_cd)
            iconv_close(cd->sb_cd);
        if (cd->db_cd)
            iconv_close(cd->db_cd);
        free(cd);
    }else{
        errno = EBADF;
    }
}

/*
** The init subroutine to be invoked when iconv_open() is called.
*/
static _LC_host_iconv_t   *init(_LC_core_iconv_t *core_cd,
               char* toname, char* fromname)
{
        _LC_host_iconv_t*  cd;
        int   i;

    for (i = 0; 1; i++) {
        if (!_iconv_host[i].local)
            return NULL;
        if (strcmp(toname, _iconv_host[i].local) == 0 &&
            strcmp(fromname, _iconv_host[i].host) == 0)
            break;
    }

        if (!(cd = (_LC_host_iconv_t *)
                       malloc(sizeof(_LC_host_iconv_t))))
                return (NULL);

    if (!(cd->sb_cd = iconv_open(toname, _iconv_host[i].sbcs))) {
        free(cd);
        return NULL;
    }
    if (!(cd->db_cd = iconv_open(toname, _iconv_host[i].dbcs))) {
        iconv_close(cd->sb_cd);
        free(cd);
        return NULL;
    }
            cd->core = *core_cd;
    cd->cntl = _iconv_host[i].fcntl;
    cd->curcd = cd->sb_cd;
    return cd;
}

/*
** The instantiate() method is called when iconv_open() loads the
** converter by a call to __lc_load().
*/
_LC_core_iconv_t         *instantiate(void)
{
        static _LC_core_iconv_t
cd;

        cd.hdr.magic = _LC_MAGIC;
        cd.hdr.version = _LC_VERSION;
        cd.hdr.type_id = _LC_ICONV;
        cd.hdr.size = sizeof (_LC_core_iconv_t);
        cd.init = init;
```

```
        cd.exec = _iconv_exec;
        cd.close = _iconv_close;
        return &cd;
}
```

Examples

1. The following example provides sample code for a stateless converter that performs an algorithm-based convertion of the IBM-850 code set to the ISO8859-1 code set. The file name for this example is 850_88591.c.

```
#include <stdlib.h>
#include <iconv.h>
#include "850_88591.h"

#define DONE     0

/*
 *  Name :  _iconv_exec()
 *
 *  This contains actual conversion method.
 */
static size_t  _iconv_exec(_LC_sample_iconv_t *cd,
      unsigned char** inbuf, size_t *inbytesleft,
      unsigned char** outbuf, size_t *outbytesleft)
/*
 *  cd      : converter descriptor
 *  inbuf       : input buffer
 *  outbuf      : output buffer
 *  inbytesleft   : number of data(in bytes) in input buffer
 *  outbytesleft    : number of data(in bytes) in output buffer
 */
{
        unsigned char   *in;   /* point the input buffer */
        unsigned char   *out;  /* point the output buffer */
        unsigned char   *e_in; /* point the end of input buffer*/
        unsigned char   *e_out; /* point the end of output                                                    buffer*/

    /*
     *   If given converter discripter is invalid,
     *   it sets the errno and returns the number
     *   of bytes left to be converted.
     */
        if (!cd) {
        errno = EBADF;
        return *inbytesleft;
        }

        /*
         *      If the input buffer does not exist or there
         *      is no character to be converted, it returns
         *      0 (no characters to be converted).
         */
        if (!inbuf || !(*inbytesleft))
                return DONE;

        /*
         *      Set up pointers and initialize other variables
         */
        e_in = (in = *inbuf) + *inbytesleft;
        e_out = (out = *outbuf) + *outbytesleft;

     /*
      *     Perform code point conversion until all input
      *     is consumed.
      *     When error occurs (i.e. buffer overflow), error
      *     number is set and exit this loop.
```

```
      */
         while (in < e_in) {

        /*
         *   If there is not enough space left in output buffer
         *   to hold the converted data, it stops converting and
         *   sets the errno to E2BIG.
         */
        if (e_out <= out) {
            errno = E2BIG;
            break;
        }

        /*
         *   Convert the input data and store it into the output
         *   buffer, then advance the pointers which point to the
         *   buffers.
         */
        *out++ = table[*in++];

    }   /* while */

    /*
     *  Update the pointers to the buffers and
     *  input /output byte counts
     */
     *inbuf = in;
        *outbuf = out;
    *inbytesleft = e_in - in;
    *outbytesleft = e_out - out;

    /*
     *   Reurn the number of bytes left to be converted
     *   (0 for successful conversion completion)
     */
    return *inbytesleft;
}

/*
 *  Name :  _iconv_close()
 *
 *  Free the allocated converter descriptor
 */
static void _iconv_close(iconv_t cd)
{
    if (!cd)
        free(cd);
    else
        /*
         *  If given converter is not valid,
         *  it sets the errno to EBADF
         */
        errno = EBADF;
}

/*
 *  Name :  init()
 *
 *  This allocates and initializes the converter descriptor.
 */
static _LC_sample_iconv_t*
init (_LC_core_iconv_t *core_cd,  char* toname, char* fromname)
{
        _LC_sample_iconv_t *cd; /* converter descriptor */

    /*
     *   Allocate a converter descriptor
```

```
     */
        if (!(cd = (_LC_sample_iconv_t *)
                        malloc(sizeof(_LC_sample_iconv_t))))
                return (NULL);

    /*
     *Copy the core part of converter descriptor which is passed       *in
     */
        cd->core = *core_cd;

    /*
     *   Return the converter descriptor
     */
        return cd;
}

/*
 *  Name :  instantiate()
 *
 *  Core part of a converter descriptor is initialized here.
 */
_LC_core_iconv_t*   instantiate(void)
{
        static _LC_core_iconv_t  cd;

    /*
     *  Initialize
     *  _LC_MAGIC and _LC_VERSION are defined in <lc_core.h>.
     *  _LC_ICONV and _LC_core_iconv_t are defined in <iconv.h>.
     */
    cd.hdr.magic = _LC_MAGIC;
    cd.hdr.version = _LC_VERSION;
    cd.hdr.type_id = _LC_ICONV;
    cd.hdr.size = sizeof (_LC_core_iconv_t);

    /*
     *  Set pointers to each method.
     */
        cd.init = init;
        cd.exec = _iconv_exec;
        cd.close = _iconv_close;

    /*
     *  Returns the core part
     */
        return &cd;
}
```

2. The following example contains a sample header file named `850_88591.h`.

```
#ifndef _ICONV_SAMPLE_H
#define _ICONV_SAMPLE_H

/*
 *      Define _LC_sample_iconv_t
 */
typedef struct _LC_sample_iconv_rec {
        _LC_core_iconv_t   core;
} _LC_sample_iconv_t;

static unsigned char     table[] = { /*


        ____________________________________________
       |                                            |
       |      IBM-850              ISO8859-1        |
       |____________________________________________|
       /*      0x00     */             0x00,
```

```
/*      0x01    */              0x01,
/*      0x02    */              0x02,
/*      0x03    */              0x03,
/*      0x04    */              0x04,
/*      0x05    */              0x05,
/*      0x06    */              0x06,
/*      0x07    */              0x07,
/*      0x08    */              0x08,
/*      0x09    */              0x09,
/*      0x0A    */              0x0A,
/*      0x0B    */              0x0B,
/*      0x0C    */              0x0C,
/*      0x0D    */              0x0D,
/*      0x0E    */              0x0E,
/*      0x0F    */              0x0F,
/*      0x10    */              0x10,
/*      0x11    */              0x11,
/*      0x12    */              0x12,
/*      0x13    */              0x13,
/*      0x14    */              0x14,
/*      0x15    */              0x15,
/*      0x16    */              0x16,
/*      0x17    */              0x17,
/*      0x18    */              0x18,
/*      0x19    */              0x19,
/*      0x1A    */              0x1A,
/*      0x1B    */              0x1B,
/*      0x1C    */              0x1C,
/*      0x1D    */              0x1D,
/*      0x1E    */              0x1E,
/*      0x1F    */              0x1F,
/*      0x20    */              0x20,
/*      0x21    */              0x21,
/*      0x22    */              0x22,
/*      0x23    */              0x23,
/*      0x24    */              0x24,
/*      0x25    */              0x25,
/*      0x26    */              0x26,
/*      0x27    */              0x27,
/*      0x28    */              0x28,
/*      0x29    */              0x29,
/*      0x2A    */              0x2A,
/*      0x2B    */              0x2B,
/*      0x2C    */              0x2C,
/*      0x2D    */              0x2D,
/*      0x2E    */              0x2E,
/*      0x2F    */              0x2F,
/*      0x30    */              0x30,
/*      0x31    */              0x31,
/*      0x32    */              0x32,
/*      0x33    */              0x33,
/*      0x34    */              0x34,
/*      0x35    */              0x35,
/*      0x36    */              0x36,
/*      0x37    */              0x37,
/*      0x38    */              0x38,
/*      0x39    */              0x39,
/*      0x3A    */              0x3A,
/*      0x3B    */              0x3B,
/*      0x3C    */              0x3C,
/*      0x3D    */              0x3D,
/*      0x3E    */              0x3E,
/*      0x3F    */              0x3F,
/*      0x40    */              0x40,
/*      0x41    */              0x41,
/*      0x42    */              0x42,
/*      0x43    */              0x43,
```

```
/*      0x44    */              0x44,
/*      0x45    */              0x45,
/*      0x46    */              0x46,
/*      0x47    */              0x47,
/*      Ox48    */              0x48,
/*      0x49    */              0x49,
/*      0x4A    */              0x4A,
/*      0x4B    */              0x4B,
/*      0x4C    */              0x4C,
/*      0x4D    */              0x4D,
/*      0x4E    */              0x4E,
/*      0x4F    */              0x4F,
/*      0x50    */              0x50,
/*      0x51    */              0x51,
/*      0x52    */              0x52,
/*      0x53    */              0x53,
/*      0x54    */              0x54,
/*      0x55    */              0x55,
/*      0x56    */              0x56,
/*      0x57    */              0x57,
/*      0x58    */              0x58,
/*      0x59    */              0x59,
/*      0x5A    */              0x5A,
/*      0x5B    */              0x5B,
/*      0x5C    */              0x5C,
/*      0x5D    */              0x5D,
/*      0x5E    */              0x5E,
/*      0x5F    */              0x5F,
/*      0x60    */              0x60,
/*      0x61    */              0x61,
/*      0x62    */              0x62,
/*      0x63    */              0x63,
/*      0x64    */              0x64,
/*      0x65    */              0x65,
/*      0x66    */              0x66,
/*      0x67    */              0x67,
/*      0x68    */              0x68,
/*      0x69    */              0x69,
/*      0x6A    */              0x6A,
/*      0x6B    */              0x6B,
/*      0x6C    */              0x6C,
/*      0x6D    */              0x6D,
/*      0x6E    */              0x6E,
/*      0x6F    */              0x6F,
/*      0x70    */              0x70,
/*      0x71    */              0x71,
/*      0x72    */              0x72,
/*      0x73    */              0x73,
/*      0x74    */              0x74,
/*      0x75    */              0x75,
/*      0x76    */              0x76,
/*      0x77    */              0x77,
/*      0x78    */              0x78,
/*      0x79    */              0x79,
/*      0x7A    */              0x7A,
/*      0x7B    */              0x7B,
/*      0x7C    */              0x7C,
/*      0x7D    */              0x7D,
/*      0x7E    */              0x7E,
/*      0x7F    */              0x7F,
/*      0x80    */              0xC7,
/*      0x81    */              0xFC,
/*      0x82    */              0xE9,
/*      0x83    */              0xE2,
/*      0x84    */              0xE4,
/*      0x85    */              0xE0,
/*      0x86    */              0xE5,
```

```
/*      0x87    */              0xE7,
/*      0x88    */              0xEA,
/*      0x89    */              0xEB,
/*      0x8A    */              0xE8,
/*      0x8B    */              0xEF,
/*      0x8C    */              0xEE,
/*      0x8D    */              0xEC,
/*      0x8E    */              0xC4,
/*      0x8F    */              0xC5,
/*      0x90    */              0xC9,
/*      0x91    */              0xE6,
/*      0x92    */              0xC6,
/*      0x93    */              0xF4,
/*      0x94    */              0xF6,
/*      0x95    */              0xF2,
/*      0x96    */              0xFB,
/*      0x97    */              0xF9,
/*      0x98    */              0xFF,
/*      0x99    */              0xD6,
/*      0x9A    */              0xDC,
/*      0x9B    */              0xF8,
/*      0x9C    */              0xA3,
/*      0x9D    */              0xD8,
/*      0x9E    */              0xD7,
/*      0x9F    */              0x1A,
/*      0xA0    */              0xE1,
/*      0xA1    */              0xED,
/*      0xA2    */              0xF3,
/*      0xA3    */              0xFA,
/*      0xA4    */              0xF1,
/*      0xA5    */              0xD1,
/*      0xA6    */              0xAA,
/*      0xA7    */              0xBA,
/*      0xA8    */              0xBF,
/*      0xA9    */              0xAE,
/*      0xAA    */              0xAC,
/*      0xAB    */              0xBD,
/*      0xAC    */              0xBC,
/*      0xAD    */              0xA1,
/*      0xAE    */              0xAB,
/*      0xAF    */              0xBB,
/*      0xB0    */              0x1A,
/*      0xB1    */              0x1A,
/*      0xB2    */              0x1A,
/*      0xB3    */              0x1A,
/*      0xB4    */              0x1A,
/*      0xB5    */              0xC1,
/*      0xB6    */              0xC2,
/*      0xB7    */              0xC0,
/*      0xB8    */              0xA9,
/*      0xB9    */              0x1A,
/*      0xBA    */              0x1A,
/*      0xBB    */              0x1A,
/*      0xBC    */              0x1A,
/*      0xBD    */              0xA2,
/*      0xBE    */              0xA5,
/*      0xBF    */              0x1A,
/*      0xC0    */              0x1A,
/*      0xC1    */              0x1A,
/*      0xC2    */              0x1A,
/*      0xC3    */              0x1A,
/*      0xC4    */              0x1A,
/*      0xC5    */              0x1A,
/*      0xC6    */              0xE3,
/*      0xC7    */              0xC3,
/*      0xC8    */              0x1A,
/*      0xC9    */              0x1A,
```

```
        /*      0xCA    */              0x1A,
        /*      0xCB    */              0x1A,
        /*      0xCC    */              0x1A,
        /*      0xCD    */              0x1A,
        /*      0xCE    */              0x1A,
        /*      0xCF    */              0xA4,
        /*      0xD0    */              0xF0,
        /*      0xD1    */              0xD0,
        /*      0xD2    */              0xCA,
        /*      0xD3    */              0xCB,
        /*      0xD4    */              0xC8,
        /*      0xD5    */              0x1A,
        /*      0xD6    */              0xCD,
        /*      0xD7    */              0xCE,
        /*      0xD8    */              0xCF,
        /*      0xD9    */              0x1A,
        /*      0xDA    */              0x1A,
        /*      0xDB    */              0x1A,
        /*      0xDC    */              0x1A,
        /*      0xDD    */              0xA6,
        /*      0xDE    */              0xCC,
        /*      0xDF    */              0x1A,
        /*      0xE0    */              0xD3,
        /*      0xE1    */              0xDF,
        /*      0xE2    */              0xD4,
        /*      0xE3    */              0xD2,
        /*      0xE4    */              0xF5,
        /*      0xE5    */              0xD5,
        /*      0xE6    */              0xB5,
        /*      0xE7    */              0xFE,
        /*      0xE8    */              0xDE,
        /*      0xE9    */              0xDA,
        /*      0xEA    */              0xDB,
        /*      0xEB    */              0xD9,
        /*      0xEC    */              0xFD,
        /*      0xED    */              0xDD,
        /*      0xEE    */              0xAF,
        /*      0xEF    */              0xB4,
        /*      0xF0    */              0xAD,
        /*      0xF1    */              0xB1,
        /*      0xF2    */              0x1A,
        /*      0xF3    */              0xBE,
        /*      0xF4    */              0xB6,
        /*      0xF5    */              0xA7,
        /*      0xF6    */              0xF7,
        /*      0xF7    */              0xB8,
        /*      0xF8    */              0xB0,
        /*      0xF9    */              0xA8,
        /*      0xFA    */              0xB7,
        /*      0xFB    */              0xB9,
        /*      0xFC    */              0xB3,
        /*      0xFD    */              0xB2,
        /*      0xFE    */              0x1A,
        /*      0xFF    */              0xA0,
};
#endif
```

3. The following example is a sample makefile.

```
SHELL   = /bin/ksh
CFLAGS  = $(COMPOPT) $(INCLUDE) $(DEFINES)
INCLUDE = -I.
COMPOPT =
DEFINES = -D_POSIX_SOURCE -D_XOPEN_SOURCE
CC      = /bin/xlc
LD      = /bin/ld
RM      = /bin/rm
```

```
SRC     = 850_88591.c
TARGET  = 850_88591

ENTRY_POINT       = instantiate

$(TARGET) :
       cc -e $(ENTRY_POINT) -o $(TARGET) $(SRC) -l iconv

clean :
        $(RM) -f $(TARGET)
        $(RM) -f *.o
```

Input Method Overview

For an application to run in the international environment for which National Language Support (NLS) provides a base, input methods are needed. The Input Method is an application programming interface (API) that allows you to develop applications independent of a particular language, keyboard, or code set. Each type of input method has the following features:

Keymaps Set of input method keymaps (imkeymaps) that works with the input method and determines the supported locales.

Keysyms Set of key symbols (keysyms) that the input method can handle.

Modifiers Set of modifiers or states, each having a mask value, that the input method supports.

See the following for more information:

- "Input Method Introduction"
- "Programming Input Methods" on page 454
- "Working with Keyboard Mapping" on page 457
- "Using Callbacks" on page 458
- "Bidirectional Input Method" on page 461
- "Cyrillic Input Method (CIM)" on page 462
- "Greek Input Method (GIM)" on page 463
- "Japanese Input Method (JIM)" on page 464
- "Korean Input Method (KIM)" on page 470
- "Latvian Input Method (LVIM)" on page 472
- "Lithuanian Input Method (LTIM)" on page 472
- "Thai Input Method (THIM)" on page 472
- "Vietnamese Input Method (VNIM)" on page 472
- "Simplified Chinese Input Method (ZIM)" on page 473
- "Simplified Chinese Input Method (ZIM-UCS)" on page 474
- "Single-Byte Input Method" on page 475
- "Traditional Chinese Input Method (TIM)" on page 477
- "Universal Input Method" on page 478
- "List of Reserved Keysyms" on page 479

Input Method Introduction

An input method is a set of functions that translates key strokes into character strings in the code set specified by your locale. Input method functions include locale-specific input processing and keyboard controls (for example, Ctrl, Alt, Shift, Lock, and Alt-Graphic). The input method allows various types of input, but only keyboard events are dealt with here.

Your locale determines which input method should be loaded, how the input method runs, and which devices are used. The input method then defines states and their outcome.

When the input method translates a keystroke into a character string, the translation process takes into account the keyboard and the code set you are using. You may want to write your own input method if you do not have a standard keyboard or if you customize your code set.

Many languages use a small set of symbols or letters to form words. To enter text with a keyboard, you press keys that correspond to symbols of the alphabet. When a character in your alphabet does not exist on the keyboard, you must press a combination of keys. Input methods provide algorithms that allow you to compose such characters.

Some languages use an ideographic writing system. They use a unique symbol, rather than a group of letters, to represent a word. For instance, the character sets used in China, Japan, Korea, and Taiwan have more than 5,000 characters. Consequently, more than one byte must be used to represent a character. Moreover, a single keyboard cannot include all the required ideographic symbols. You need input methods that can compose multibyte characters.

The **/usr/lib/nls/loc** directory contains the input methods installed on your system. You can list the contents of this directory to determine which input methods are available to you. Input method file names have the format *Language_Territory*.**im**. For example, the **fr_BE.im** file is the input method file for the French language as used in Belgium.

Through a well-structured protocol, input methods allow applications to support different input without using locale-specific input processing.

In AIX, the input method is provided in the aixterm. When characters typed from the AIXwindows interface reach the server, the characters are in the form of key codes. These key codes are converted into keysyms as defined by the table provided in the client. This table contains mappings for each of the key codes into a predefined set of codes called the keysyms. Any key code generated by a keyboard should have a keysym. These keysyms are maintained and allocated by the MIT X Consortium. The keysyms are passed to the client aixterm terminal emulator. In the aixterm, the input keysyms are converted into file codes by the input method and are then sent to the application. The X server is designed to work with the display adapter provided in the system hardware. The X server communicates with the X client through sockets. Thus, the server and the client can reside on different systems in a network, provided they can communicate with each other. The data from the keyboard enters the X server, and from the server it gets passed to the terminal emulator. The terminal emulator passes the data to the application. When data comes from applications to the display device, it passes through the terminal emulator by sockets to the server and from the server to the display device.

Input Method Names

The set of input methods available depends on which locales have been installed and what input methods those locales provide. The name of the input method usually corresponds to the locale. For example, the Greek Input Method is named el_GR, which is the same as the locale for the Greek language spoken in Greece.

When there is more than one input method for a locale, any secondary input method is identified by a modifier that is part of the locale name. For example, there are two input methods for the French locale as spoken in Canada, the default and an alternative method that supports the earlier keyboard. The input method names are:

fr_CA	Default input method
fr_CA@im=alt	Alternative input method
fr_CA.im__64	64-bit input method

The **fr** portion of the locale represents the language name (French), and the **CA** represents the territory name (Canada). The **@im=alt** string is the modifier portion of the locale that is used to identify the alternative input method. All modifier strings are identified by the format **@im=***Modifier*.

Since the input method is a loadable object module, a different object is required when running in the 64-bit environment. In the 64-bit environment, the input method library will automatically append **__64** to the name when searching for the input method. In the example above, the name of the input method would be **fr_CA.im__64**.

It is possible to name input methods without using the locale name. Since the **libIM** library does not restrict names to locale names, the calling application must ensure that the name passed to **libIM** can be found. However, applications should request only modifier strings of the form **@im=***Modifier* and that the user's request be concatenated with the return string from the **setlocale** (**LC_CTYPE,NULL**) subroutine.

Input Method Areas

In order to compose, complex input methods require direct dialog with users. For example, the Japanese Input Method may need to show a menu of candidate strings based on the phonetic matches of the keys you enter. The feedback of the key strokes appears in one or more areas on the display. The input method areas are:

Status
: Text data and bitmaps can appear in the Status area. The Status area is an extension of the light-emitting diodes (LEDs) on the keyboard.

Pre-edit
: Intermediate text appears in the Pre-edit area for languages that compose before the client handles the data.

  A common feature of input methods is that you press a combination of keys to represent a single character or set of characters. This process of composing characters from keystrokes is called *pre-editing*.

Auxiliary
: Pop-up menus and dialog that allow you to customize the input method appear in the Auxiliary area. You can have multiple Auxiliary areas managed by the input method and independent of the client.

  Management for input method areas is based on the division of responsibility between the application (or toolkit) and the input method. The divisions of responsibility are:

  - Applications are responsible for the size and position of the input method area.
  - Input methods are responsible for the contents of the input area. The input method area cannot suggest a placement.

Related Information

"Chapter 16. National Language Support" on page 329.

"Locale Subroutines" on page 340.

Programming Input Methods

The input method is a programming interface that allows applications to run in an international environment provided through National Language Support (NLS). The input method has the following characteristics:

- Localized input support (defined by locale)
- Multiple keyboard support

- Multibyte character-input processing

Initialization

You can use the **IMQueryLanguage** subroutine to determine if an input method is available without initializing it. An application (toolkit) initializes a locale-specific input method by calling the **IMInitialize** subroutine, which initializes a locale-specific input method editor (IMED). The subroutine uses the **LOCPATH** environment variable to search for the input method named by the **LANG** environment variable. The **LOCPATH** variable specifies a set of directory names used to search for input methods.

If the input method is found, the **IMInitialize** subroutine uses the **load** subroutine to load the input method and attach the **imkeymap** file. When the input method is accessed, an object of the type **IMFep** (input method front-end processor) is returned. The **IMFep** should be treated as an opaque structure.

The **IMInitialize** subroutine links the converter function using the **load** subroutine. The **load** subroutine is similar to the **exec** subroutine and links the converter program at run-time. Since the **IMInitialize** subroutine is called as a library function, it must preserve security for certain programs. When the **IMInitialize** subroutine is called from a set root ID program, it will ignore the **LOCPATH** environment variable and search for converters only in the **/usr/lib/nls/loc/iconv** and **/etc/nls/loc/iconv** directories.

Each **IMFep** inherits the locale's code set when the **IMInitialize** subroutine is called. Consequently, strings returned by the **IMFilter** and **IMLookupString** subroutines are in the locale's code set. Changing the locale after the **IMInitialize** subroutine is called does not affect the code set of the **IMFep**.

For each **IMFep**, the application can use the **IMCreate** subroutine to create one or more **IMObject** instances. The **IMObject** manages its own state and can manage several Input Method Areas (see "Input Method Areas" on page 454). How each **IMObject** defines input processing depends on the code set and keyboard associated with the locale. In the simplest case, a single **IMObject** is needed if the application is managing a single dialog with the user. The input method also supports newer user interfaces where the application allows multiple dialogs with the user, and each dialog requires one **IMObject**.

The difference between an **IMFep** and **IMObject** is that the **IMFep** is a handle that binds the application to the code of the input method, while the **IMObject** is a handle that represents an instance of a state of an input device, such as a keyboard. The **IMFep** does not represent a state of the input method. Each **IMObject** is initialized to a specific input state and is changed according to the sequence of events it receives.

Once the **IMObject** is created, the application can process key events. The application should pass key events to the **IMObject** using the **IMFilter** and **IMLookupString** subroutines. These subroutines are provided to isolate the internal processing of the IMED from the customized key event mapping process.

Input Method Management

The input method provides the following subroutines for maintenance purposes:

IMInitialize	Initializes the standard input method for a specified language. Returns a handle to an IMED associated with the locale. The handle is an opaque structure of type **IMFep**.
IMQueryLanguage	Checks whether the specified language is supported.
IMCreate	Creates one instance of a particular input method. This subroutine must be called before any key event processing is performed.
IMClose	Closes the input method.
IMDestroy	Destroys an instance of an input method.

IM Keymap Management

The input method provides several subroutines to map key events to a string. The mapping is maintained in an imkeymap file located in the **LOCPATH** directory. The subroutines used for mapping are:

IMInitializeKeymap	Initializes the imkeymap associated with a specified language.
IMFreeKeymap	Frees resources allocated by the **IMInitializeKeymap** subroutine.
IMAIXMapping	Translates a pair of key-symbol and state parameters to a string and returns a pointer to that string.
IMSimpleMapping	Translates a pair of key-symbol and state parameters to a string and returns a pointer to that string.

Key Event Processing

Input processing begins when you press keys on the keyboard. The application must have created an **IMObject** before calling these functions:

IMFilter	Asks the IMED to indicate if a key event is used internally. If the IMED is composing a localized string, it maps the key event to that string.
IMLookupString	Maps the key event to a localized string.
IMProcessAuxiliary	Notifies the input method of input for an auxiliary area.
IMIoctl	Performs a variety of control or query operations on the input method.

Callbacks

The IMED communicates directly with the user by using the Input Method-Callback (IM-CB) API to access the graphic-dependent functions (callbacks) provided by the application. The application attaches the callbacks, which perform output functions and query information, to the **IMObject** during initialization. The application still handles all the input.

The set of callback functions that the IMED uses to communicate with a user must be provided by the caller. See “Using Callbacks” on page 458 for a discussion of the subroutines defined by the IM-CB API.

Input Method Structures

The major structures used by the input method are:

IMFepRec	Contains the front end information.
IMObjectRec	Contains the common part of input method objects.
IMCallback	Registers callback subroutines to the **IMFep**.
IMTextInfo	Contains information about the text area, primarily the pre-editing string.
IMAuxInfo	Defines the contents of the auxiliary area and the type of processing requested.
IMIndicatorInfo	Indicates the current value of the indicators.
IMSTR	Designates strings that are not null-terminated.
IMSTRATT	Designates strings that are not null-terminated and their attributes.

Working with Keyboard Mapping

The following model shows how input methods are used by applications. It can help you understand how to customize keyboard mapping.

Input processing is divided into three steps:

1. keycode/keystate(raw) - > keysym/modifier(new)

   This step is application and environment-dependent. The application is responsible for mapping the raw key event into a keysym/modifier for input to the input method.

   In the AIXwindows environment, the client uses the server's keysym table, **xmodmap**, which is installed at the server, to perform this step. The **xmodmap** defines the mapping of the Shift, Lock, and Alt-Graphic keys. The client uses the **xmodmap** as well as the Shift and Lock modifiers from the X event to determine the keysym/modifier represented by this event.

   For example, if you press the `XK_a` keysym with a Shift modifier, the **xmodmap** maps it to the `XK_A` keysym. Since you used the Shift key to map the key code to a keysym, the application should mask the Shift modifier from the original X event. Consequently, the input to the input method (step 2) would be the `XK_A` keysym and no modifier.

   In another environment, if the device provides no additional information, the input method receives the `XK_a` keysym with the Shift modifier. The input method should perform the same mapping in both cases and return the letter A.

2. keysym/modifier(new) - > localized string

   This step depends on the localized IMED and varies with each locale. It is used to notify the IMED that a key event occurred and to ask for an indication that their IMED uses the key event internally. This occurs when the application calls the **IMFilter** subroutine.

   If the IMED indicates that the key event is used for internal processing, the application ignores the event. Since the IMED is the first to see the event, this step should be done before the application interprets the event. The IMED only uses key events that are essential.

   If the IMED indicates the event is not used for internal processing, the application performs the next step.

3. keysym/modifier(new) - > customized string

   This step occurs when the application calls the **IMLookupString** subroutine. The input method keymap (created by the **keycomp** command) defines the mapping for this phase. It is the last attempt to map the key event to a string and allows a user to customize the mapping.

   If the keysym/modifier (new) combination is defined in the input method keymap (imkeymap), a string is returned. Otherwise, the key event is unknown to the input method.

IM Keymaps

The input method provides support for user-defined imkeymaps, allowing you to customize input method mapping. The input methods support imkeymaps for each locale. The file name for imkeymaps is similar to that of input methods, except that the suffix for imkeymap files is **.imkeymap** instead of **.im**.

Refer to this example of using the Italian input method, which illustrates how you can customize your imkeymap:

1. To copy the default imkeymap source file to your **$HOME** directory, enter:

   ```
   cd $HOME
   cp /usr/lib/nls/loc/it_IT.ISO8859-1.imkeymap.src .
   ```

2. To edit the imkeymap source file following the default file format, enter:

   ```
   vi it_IT.ISO8859-1.imkeymap.src
   ```

3. To compile the imkeymap source file, enter:

   ```
   keycomp < it_IT.ISO8859-1.imkeymap.src > it_IT.ISO8859-1.imkeymap
   ```

4. To make sure the **LOCPATH** variable specifies **$HOME** before /**usr/lib/nls/loc**, enter:
   ```
   LOCPATH=$HOME:$LOCPATH
   ```

   Note: All **setuid** and **setgid** programs will ignore the **LOCPATH** environment variable.

Inbound and Outbound Mapping

The imkeymaps map a key symbol to a file code set string. The localized imkeymaps found in the **/usr/lib/nls/loc** library are defined to include mapping for all of the inbound keys. The imkeymaps provide two types of mapping:

Inbound mapping	Mapping of a keysym/modifier that generates a target string encoded in the code set of the locale.
Outbound mapping	Mapping of a keysym/modifier that does not generate a target string included in the code set of the locale.

A special imkeymap, /**usr**/**lib**/**nls**/**loc**/**C@outbound.imkeymap**, defines outbound mapping for all keyboards made by this manufacturer and is primarily intended for use by aixterm. This imkeymap includes mapping of PF keys, cursor keys, and other special keys commonly used by applications. Internationalized applications that use standard input and standard output should limit their dependency on outbound mapping, which does not vary on different keyboards. For example, the Alt-a is defined in the same way on all keyboards made by this manufacturer. Yet, the Alt-tilde is different depending on the keyboard used.

The aixterm bases its outbound mapping on the **C@outbound** imkeymap. Applications that require more mapping should modify the localized imkeymap source to include the necessary definitions.

Using Callbacks

Applications that use input methods should provide callback functions so that the Input Method Editior (IMED) can communicate with the user. The type of input method you use determines whether or not callbacks are necessary. For example, the single-byte input method does not need callbacks, but the Japanese input method uses them extensively with the pre-edit facility. Pre-editing allows processing of characters before they are committed to the application.

When you use an input method, only the application can insert or delete pre-edit data and scroll the text. Consequently, the echo of the keystrokes is achieved by the application at the request of the input method logic through callbacks.

When you enter a keystroke, the application calls the **IMFilter** subroutine. Before returning, the input method can call the echoing callback function for inserting new keystrokes. Once a character has been composed, the **IMFilter** subroutine returns it, and the key strokes are deleted.

In several cases, the input method logic has to call back the client. Each of these is defined by a callback action. The client specifies what callback should be called for each action.

There are three types of callbacks:

- Text drawing

  The IMED uses text callbacks to draw any pre-editing text currently being composed. When the callbacks are needed, the application and the IMED share a single-line buffer, where the editing is performed. The IMED also provides cursor information that the callbacks then present to the user.

  The text callbacks are:

IMTextDraw	Asks the application program to draw the text string.

IMTextHide Tells the application program to hide the text area.
IMTextStart Notifies the application program of the length of the pre-editing space.
IMTextCursor Asks the application program to move the text cursor.

- Indicator (status)

  The IMED uses indicator callbacks to request internal status. The **IMIoctl** subroutine works with the **IMQueryIndicatorString** command to retrieve the text string that tells the internal status. Indicator callbacks are similar to text callbacks, except that instead of sharing a single-line buffer, a status value is used.

  The Indicator callbacks are:

IMIndicatorDraw Tells the application program to draw the status indicator.
IMIndicatorHide Tells the application program to hide the status indicator.
IMBeep Tells the application program to emit a beep sound.

- Auxiliary

  The IMED uses auxiliary callbacks to request complex dialogs with the user. Consequently, these callbacks are more sophisticated than text or status callbacks.

  The Auxiliary callbacks are:

IMAuxCreate Tells the application program to create an auxiliary area.
IMAuxDraw Tells the application program to draw an auxiliary area.
IMAuxHide Tells the application program to hide an auxiliary area.
IMAuxDestroy Tells the application program to destroy an auxiliary area.

The **IMAuxInfo** structure defines the dialog needed by the IMED.

The contents of the auxiliary area are defined by the **IMAuxInfo** structure, found in the **/usr/include/im.h** library.

The **IMAuxInfo** structure contains six fields.

IMTitle Defines the title of the auxiliary area. This is a multibyte string. If `title.len` is 0, there is no title to be displayed.

IMMessage Defines a list of messages to be presented. From the applications perspective, the **IMMessage** structure should be treated as informative, output-only text. However, some input methods use the **IMMessage** structure to conduct a dialog with the user in which the key events received by way of the **IMFilter** or **IMLookupString** subroutine are treated as input to the input method. In such cases, the input method may treat the **IMMessage** structure as either a selectable list or a prompt area. In either case, the application displays only the message contents.

The **IMProcessAuxiliary** subroutine need not be called if the **IMSelection** structure contains no **IMPanel** structures and the `IMButton` field is null.

The `message.nline` indicates the number of messages contained in the **IMMessage** structure. Each message is assumed to be a single line. Control characters, such as \t, are not recognized. The text of each message is defined by the **IMSTRATT** structure, which consists of both a multibyte string and an attribute string. Each attribute is mapped one-to-one for each byte in the text string.

If `message.cursor` is True, then the **IMMessage** structure defines a text cursor at location `message.cur_row`, `message.cur_col`. The `message.cur_col` field is defined in terms of bytes. The `message.maxwidth` field contains the maximum width of all text messages defined in terms of columns.

IMButton Indicates the possible buttons that can be presented to a user. The `IMButton` field tells the application which user interface controls should be provided for the end user. The button member is of type int and may contain the following masks:

IM_OK Present the OK button.

IM_CANCEL
Present the CANCEL button.

IM_ENTER
Present the ENTER button.

IM_RETRY
Present the RETRY button.

IM_ABORT
Present the ABORT button.

IM_YES
Present the YES button.

IM_NO Present the NO button.

IM_HELP
Present the HELP button.

IM_PREV
Present the PREV button.

IM_NEXT
Present the NEXT button.

The application should use the **IMProcessAuxiliary** subroutine to communicate the button selection.

IMSelection Defines a list of items, such as ideographs, that an end user can select. This structure is used when the input method wants to display a large number of items but does not want to control how the list is presented to the user.

The **IMSelection** structure is defined as a list of **IMPanel** structures. Not all applications support **IMSelection** structures inside the **IMAuxInfo** structure. Applications that do support **IMSelection** structures should perform the **IM_SupportSelection** operation using the **IMIoctl** subroutine immediately after creation of the **IMObject**. In addition, not all applications support multiple **IMPanel** structures. Therefore, the `panel_row` and `panel_col` fields are restricted to a setting of 1 by all input methods.

Each **IMPanel** structure consists of a list of `IMItem` fields that should be treated as a two-dimensional, row/column list whose dimensions are defined as `item_row` times `item_col`. If `item_col` is 1, there is only one column. The size of the **IMPanel** structure is defined in terms of bytes. Each item within the **IMPanel** structure is less than or equal to `panel->maxwidth`.

The application should use the **IMProcessAuxiliary** subroutine to communicate one or more user selections. The **IM_SELECTED** value indicates which item is selected. The **IM_CANCEL** value indicates that the user wants to terminate the auxiliary dialog.

hint Used by the input method to provide information about the context of the **IMAuxInfo** structure. A value of `IM_AtTheEvent` indicates that the **IMAuxInfo** structure is associated with the last event passed to the input method by either the **IMFilter** or **IMLookupString** subroutine. Other hints are used to distinguish when multiple **IMAuxInfo** structures are being displayed.

status Used by the input method for internal processing. This field should not be used by applications.

Each **IMAuxInfo** structure is independent of the others. The method used for displaying the members is determined by the caller of the input method. The **IMAuxInfo** structure is used by the **IMAuxDraw** callback.

Initializing Callbacks

All callbacks must be identified when you call the **IMCreate** subroutine. The IMCallback structure contains the address for each callback function. The caller of the **IMCreate** subroutine must initialize the **IMCallback** structure with the addresses.

The callback functions can be called before the **IMCreate** subroutine returns control to the caller. Usually, the **IMTextStart** callback is called to identify the size of the pre-edit buffer.

Bidirectional Input Method

The Bidirectional Input Method (BIM) is similar to the Single-Byte Input Method except that it is customized to process the Arabic and Hebrew keyboards. BIM also links the Hebrew and Arabic states to the Latin states. The Alt+Right Shift keys allow the user to toggle between the Arabic/Hebrew and Latin language layers. The use of these keys is derived from BIM. The features of BIM are:

- Supports Arabic, Hebrew, and Latin states.
- Supports the ISO8859-6, ISO8859-8, IBM-1046, and IBM-856 code sets.
- Performs diacritical composing.
- Keymaps:

  ar_AA.ISO8859-6.imkeymap

  ar_AA@alt.ISO8859-6.imkeymap

  Ar_AA.IBM-1046.imkeymap

  Ar_AA@alt.IBM-1046.imkeymap

  iw_IL.ISO8859-8.imkeymap

  iw_IL@alt.ISO8859-8.imkeymap

  Iw_IL.IBM-856.imkeymap

  Iw_IL@alt.IBM-856.imkeymap
- Key Settings:

scr-rev()	Reverses the screen orientation and sets the keyboard layer to the default language of the new orientation.
ltr-lang()	Enables the Latin keyboard layer.
rtl-lang()	Enables the Arabic/Hebrew keyboard layer.
col-mod()	Enables the column heading adjustment, which handles each word as a separate column.
auto-push()	Toggles the Autopush mode. This mode handles mixed left-to-right and right-to-left text. When you enable the Autopush mode, reversed segments are automatically initiated and terminated according to the entered character or the selected language layer. Thus, you are relieved of manually invoking the Push function.
chg-push()	Toggles the Push mode. This mode causes the cursor to remain in its position and pushes the typed characters in the direction opposed to the field direction.
shp-in()	Shapes Arabic characters in their initial forms.
shp-is()	Shapes Arabic characters in their isolated forms.
shp-p()	Shapes Arabic characters in their passthru forms.
shp-asd()	Shapes Arabic characters in their automatic forms.
shp-m()	Shapes Arabic characters in their middle forms.
shp-f()	Shapes Arabic characters in their final forms.

- Modifiers:

ShiftMask	0x01

LockMask	0x02
ControlMask	0x04
Mod1Mask (Left-Alt)	0x08
Mod2Mask (Right-Alt)	0x10

Cyrillic Input Method (CIM)

The Cyrillic Input Method (CIM) is similar to the Single-Byte Input Method, except that it is customized for processing the Cyrillic keyboard. The features of CIM are:

- Supports Cyrillic and Latin states.

  You can toggle between the two states by pressing the Alt key and the Left or Right Shift key simultaneously.

  Note: The Alt-Graphic (Right Alt) key can be used to generate additional characters within each keyboard layer.
- For the Russian and Bulgarian locales, both 101-key and 102-key keyboard drivers are supported.
- Supports the ISO8859-5 code set.

Keymap:

bg_BG.ISO8859-5.imkeymap

mk_MK.ISO8859-5.imkeymap

sr_SP.ISO8859-5.imkeymap

ru_RU.ISO8859-5.imkeymap

be-BY.ISO8859-5.imkeymap

uk-UA.ISO8859-5.imkeymap

Keysyms:

The CIM uses the keysyms in the **XK_CYRILLIC**, **XK_LATIN1**, and **XK_MISCELLANY** groups.

Reserved Keysyms:

XK_dead_acute	0x180000b4
XK_dead_grave	0x18000060
XK_dead_circumflex	0x1800005e
XK_dead_diaeresis	0x180000a8
XK_dead_tilde	0x1800007e
XK_dead_caron	0x180001b7
XK_dead_breve	0x180001a2
XK_dead_doubleacute	0x180001bd
XK_dead_degree	0x180000b0
XK_dead_abovedot	0x180001ff
XK_dead_macron	0x180000af
XK_dead_cedilla	0x180000b8
XK_dead_ogonek	0x180001b2
XK_dead_accentdieresis	0x180007ae

The preceding keysyms are unique to the input method of this system.

Modifiers

- Modifiers:

ShiftMask	0x01
LockMask	0x02
ControlMask	0x04
Mod1Mask (Left-Alt)	0x08
Mod2Mask (Right-Alt)	0x10

- Internal Modifier:

Cyrillic Layer	0x20

Related Information

Greek Input Method (GIM)

The Greek Input Method (GIM) is similar to the Single-Byte Input Method (SIM), but has been extended to handle both Latin and Greek character sets. This is accomplished by providing two layers or states of keyboard mappings, which correspond to the two character sets.

The keyboard is initially in the Latin input state. However, if the left-shift key is pressed while the left-alt key is held down, the keyboard is put in the Greek input state. The keyboard can be returned to the Latin state by pressing the right-shift key, while the left-alt key is held down. These are locking shift keys, since the state is locked when they are pressed.

While in the Greek state, the input method recognizes the following diacritical characters and valid subsequent characters for diacritical composing.

Greek Composing Characters	
Keysym	**Valid Composing Characters**
	Uppercase and Lowercase:
dead_acute	alpha, epsilon, eta, iota, omicron, upsilon, omega
dead_diaeresis	iota, upsilon
	Lowercase Only:
dead_accentdiaeresis	iota, upsilon

In the Latin state, there are no composing diacriticals, and the above keys are treated as simple graphic characters.

The Greek and Single-Byte Input Methods also differ in their handling of illegal diacritical composing sequences. In such cases, the GIM beeps and returns no characters. The SIM does not beep and returns both the diacritical character and a graphic character associated with the invalid key.

> **Note:** The Alt-Graphic (right-alt) key can be used to generate additional characters within each keyboard state.

Keymap:

el_GR.ISO8859-7.imkeymap.

Keysyms:

The GIM uses the keysyms in the **XK_LATIN1**, **XK_GREEK**, and **XK_MISCELLANY** groups.

Reserved keysyms:

XK_dead_acute	0x180000b4
XK_dead_grave	0x18000060
XK_dead_circumflex	0x1800005e
XK_dead_diaeresis	0x180000a8
XK_dead_tilde	0x1800007e
XK_dead_caron	0x180001b7
XK_dead_breve	0x180001a2
XK_dead_doubleacute	0x180001bd
XK_dead_degree	0x180000b0
XK_dead_abovedot	0x180001ff
XK_dead_macron	0x180000af
XK_dead_cedilla	0x180000b8
XK_dead_ogonek	0x180001b2
XK_dead_accentdieresis	0x180007ae

The preceding keysyms are unique to the input method of this system.

- Modifiers:

ShiftMask	0x01
LockMask	0x02
ControlMask	0x04
Mod1Mask (Left-Alt)	0x08
Mod2Mask (Right-Alt)	0x10

- Internal Modifier:

Greek Layer	0x20

Japanese Input Method (JIM)

The Japanese Input Method (JIM) is a sophisticated input method that provides Japanese input. The features include:

- Supports Romaji to Kana character conversion (RKC).
- Supports Kana to Kanji character conversion (KKC).
- Includes Hankaku (half-width) and Zenkaku (full-width) character input.
- Provides system and user dictionary lookup.
- Provides runtime registration of a word to the user dictionary.
- Requires Callback functions to support:
  - Status and Pre-edit drawing
  - All candidate menus
  - JIS Kutan number input and IBM Kanji number input

- Supports IBM-943, IBM-932 and IBM-eucJP code sets.
  For internal processing, the JIM uses the IBM-932 code set. However, it supports any code set, such as IBM-eucJP, that can be converted from IBM-932.
- Located in the **/usr/lib/nls/loc/JP.im** file.

  All other localized input methods are aliases to this file.

The Japanese code sets consist of three character groups:

- Katakana
- Hiragana
- Kanji

Katakana and Hiragana consist of about 50 characters each and form the set of phonetic characters referred to as Kana. All of the sounds in the Japanese language can be represented in Kana.

Kanji is a set of ideographs. A simple concept can be represented by a single Kanji character, while more complicated meanings can be formed with strings of Kanji characters. There are several thousand Kanji characters.

The Japanese also use the Roman alphabet. Called Romaji, the Roman alphabet consists of 26 characters. It is used mostly in technical and professional environments to represent technical vocabulary that does not exist in Japanese. A typical sentence is usually a mixture of Katakana, Hiragana, Kanji, Romaji, numbers, and other characters.

Japanese Character Processing

The Japanese Industrial Standard (JIS) specifies about 7000 Kanji characters processed by computer systems. Japanese products made by this manufacturer support all of the standard characters and more. Input of the characters is accomplished through:

- Kana-to-Kanji conversion (KKC)
- Romaji-to-Kana conversion (RKC)

The following special keys appear on the 106-key Japanese keyboard to allow for these conversions:

Special Japanese Keys		
Key Function	**Key Name**	**Description of Function**
KKC Non-conversion key	muhenkan	Leaves Kana characters as is.
KKC Conversion key	henkan	Converts Kana to Kanji.
KKC All Candidates key	zenkouho	Shows all possible Kanji representatives.
RKC Romaji Mode key	romaji	Toggles RKC on and off.
Hiragana Shift key	hiragana	Becomes Hiragana shift state.
Katakana Shift key	katakana	Becomes Katakana shift state.
Romaji Shift key	eisu	Becomes Romaji shift state.

Note: Shift states are maintained until you press another shift key. The initial state is Romaji.

Kana-To-Kanji Conversion (KKC) Technology

The Japanese Input Method's (JIM) KKC technology is based on the fact that every Kanji character or set of Kanji characters has a phonetic sound or sounds that can be expressed by Katakana or Hiragana characters.

It is much easier to input Hiragana or Katakana characters than Kanji characters. The JIM analyzes the phonetic values of the Hiragana and Katakana characters to determine the best Kanji-character equivalent. Such phonetic analysis depends on the dictionary and tables provided to the JIM.

Input Modes

The JIM has three different modes that can be used to control the input processing:

- Keyboard Mapping

  Allows invocation of alphanumeric, Katakana, or Hiragana modes.
- Character Size

  Inputs in Zenkaku (full-width) or Hankaku (half-width) mode.
- RKC off/on

  Inputs Kana directly or invoke the pre-edit composing mode to input Kana with a combination of alphabetic characters. The pre-editing facility allows processing of characters before they are committed to the application.

When the keyboard mapping mode is alphanumeric and the character size mode is Hankaku, the JIM maps keys to Romaji characters. This mode combination is known as the ″English″ mode. Pre-editing is not needed in English mode and cannot be invoked regardless of the RKC mode setting. The other mode combinations may initiate pre-editing and characters generated in these modes are not ASCII.

The following keys are used to perform Kana-to-Kanji conversion by the JIM.

Keysym	**Keyboard Mapping**
Katakana	Katakana shift
Eisu_toggle	Alphanumeric shift
Hiragana	Hiragana shift

Keysym	**Character Size**
Zenkaku_Hankaku	Full-width or Half-width toggle
Hankaku	Half-width
Zenkaku	Full-width

Keysym	**RKC on/off**
Alt-Hiragana	Enables/Disables Romaji-to-Kana conversion
Romaji	*The same effect

* Keysyms unique to the manufacturer

The following keys are also used when the JIM is pre-editing a Kanji string.

Keysym	**Kanji pre-edit**
Muhenkan	Non-conversion - commit Kana
Henkan	Conversion - get next candidate
Kanji	Same as Henkan
BunsetsuYomi	*Moves back a phrase
MaeKouko	*Moves to previous candidate
LeftDouble	*Moves cursor two characters left

RightDouble	*Moves cursor two characters right
ErInput	*Discards the current pre-edit string

Keysym	Auxiliary pre-edit
Alt-Henkan	All candidates
Touroku	Runtime registration
ZenKouho	*All candidates (the same effect)
KanjiBangou	*Kanji Number Input
HenkanMenu	*Changes conversion mode

* Keysyms unique to the manufacturer

Keyboard Mapping

There are 3 possible keyboard mapping states: Alphanumeric (Romaji), Katakana and Hiragana. Each state is invoked by a keysym that acts as a locking shift key. The keysyms are Katakana, Eisu_toggle, and Hiragana shift.

When one of these keysyms is pressed, keyboard mapping enters the state associated with the key. This state is maintained until one of the other keysyms is pressed. The initial shift state is Eisu_toggle, which can be changed by customization.

When you invoke the Hiragana or Katakana state, each key is mapped to a phonetic character within the respective character set. For example, if you press q, a Hiragana character pronounced ″ta″ is produced during Hiragana shift state, a Katakana character pronounced ″ta″ is produced during Katakana shift state, or a Romaji ″q″ is produced during Eisu_toggle shift state. On Japanese IBM keyboards, the tops of keys show all three symbols.

Also, when keyboard mapping is in Hiragana state, the input method is automatically put into a composing pre-editing mode where each Hiragana character can be converted into a Kanji character. See “Kanji Pre-edit” on page 468 for more information.

Some keys have two Hiragana or Katakana characters assigned. For example, the 7 key has large and small Hiragana characters both having the pronunciation ″ya″. These characters are not upper and lower case equivalents of each other since Kanji, Hiragana, and Katakana do not have uppercase and lowercases. The small characters are used to express special phonetic sounds. These characters can be distinguished by using the shift key.

Character Size

A subset of the Japanese character set is represented in both full-width and half-width. Kanji ideographic characters are usually full-width. The phonetic and ASCII characters have both full-width and half-width representations. The user controls character size by pressing the Zenkaku_Henkaku keysym which toggles between full-width and half-width.

Romaji-To-Kana Conversion (RKC)

For users familiar with alphanumeric keyboards, it is easier to key in the phonetic sounds rather than the Hiragana or Katakana characters. The JIM provides Romaji-to-Kana conversion (RKC), allowing the user to type in the phonetic sounds of Hiragana or Katakana characters on an alphanumeric keyboard.

Kanji Pre-edit

When operating in Romaji-To-Kana conversion mode, you must follow two steps to produce Kanji characters. First, the user inputs Hiragana characters by typing their Romaji phonetic characters. In this step, you produce a Hiragana character by typing 1 to 3 Romaji alphabetic keys that compose the phonetic sound of the Hiragana character. Second, convert the Hiragana characters to Kanji characters by pressing the Henkan key. Many Kanji characters may be associated with a single phonetic phrase. The Henkan key displays the most likely Kanji candidates. Repeated pressing of the Henkan key displays all the additional candidates.

For example, when entering the Kanji characters for the phonetic sound ″k-a-n-j-i″, you must do two things:

1. Set the keyboard mapping to the Hiragana state.
2. Enable Romaji-to-Kana mapping by pressing the Alt-Hiragana key. This action invokes the alphanumeric keyboard.

You may now press the keys that spell ″kanji″. As each phonetic sound is completed, a Hiragana character is displayed.

The Hiragana character is displayed with visual feedback to indicate that the JIM is composing in a pre-edit state. The character is underlined and shown in reverse video. This feedback facility is known as a callback. See “Using Callbacks” on page 458 for more information.

To convert the Hiragana character within the pre-edit string to a Kanji character, press the Henkan key. The most likely candidate associated with the phonetic Hiragana sound is displayed. Pressing this key repeatedly shows other candidates.

During the composition process, the pre-edit string is partitioned into segments that can be considered Kanji words. Once a string of kana characters is converted into a candidate, it is treated as one of these convertible segments. While the pre-edit string is displayed, the JIM uses the cursor key and other keys to manipulate the string.

To commit the pre-edit string to the program, the user presses the Enter key. In this case, the Enter key code itself is not sent to the program, only the string.

The Muhenkan keysym can also be used to turn off pre-edit and commit the Hiragana or Katakana character directly to the program.

The Keyboard Shift-State Transition table depicts the shift state transition and the interaction of the RKC mode key with the shift states.

Table 8.

Character Encoding	Code Points	Description	Count
000xxxxx	00–1F	Controls	32
00100000	20	Space	1
0xxxxxxx	21–7E	7–bit ASCII	94
01111111	7F	Delete	1
10000000	80	Undefined	1
100xxxxx 01xxxxxx	[81–9F] [40–7E]	Double byte	1953
100xxxxx 1xxxxxxx	[81–9F] [80–FC]	Double byte	3844
10100000	A0	Undefined	1
1xxxxxxx	A1–DF	8–bit single byte	63

Table 8. (continued)

111xxxxx 01xxxxxx	[E0–FC] [40–7E]	Double byte	1827
111xxxxx 1xxxxxxx	[E0–FC] [80–FC]	Double byte	3596
11111101	FD	Undefined	1
11111110	FE	Undefined	1
11111111	FF	All ones	1

There are 4 types of auxiliary areas within the JIM.

- All Candidates menu
- Kanji Number Input dialog
- Conversion Mode menu
- Runtime Registration dialog

A Kana-to-Kanji conversion operation on a string of Hiragana or Katakana characters can yield from one to a hundred Kanji candidates. At worst, you would have to press the conversion key more than a hundred times to get the correct Kanji character.

In such cases, it is more convenient to find the correct character by requesting the All Candidates menu with the ZenKouho or the Alt-Henkan keysym. This menu appears if the current target (a Kanji word that the cursor is pointing to in the pre-edit area) has several alternative candidates associated with it. The menu contains multiple candidates for selection. The All Candidates menu disappears when the Reset keysym is pressed, the Enter key is pressed, or a candidate is selected.

A Kanji Number Input dialog prompts the user to select the Kanji character by entering 3 to 5 digits. The digits represent the code of the character. Online dictionaries allow a user to search for the code. The ordering formats for these dictionaries vary. For example, one dictionary lists codes by phonetic sound. Another dictionary orders codes by the number of strokes used to compose the character. The KanjiBangou keysym invokes this menu. The menu is terminated with either the Reset or Return keysym.

The HenkanMenu keysym invokes the Conversion Mode menu. Four items are displayed for selection. The most important items are the word-conversion mode and phrase-conversion mode. Make a selection by choosing a number and pressing the Return keysym. This menu is terminated when either a selection is made or the Reset keysym is pressed.

A runtime registration dialog prompts the user to input a Kana string and a Kanji string for registering the mapping of the strings in the user dictionary. Once the pair is registered, the JIM can use it as a conversion candidate. The menu is terminated with the Escape or Reset keysym.

The presentation of menus depends on the interface environment in which the JIM is operating. For example, some interfaces support scrolling menus that use the Page Down and Page Up keys. Discussion of these interfaces is outside the scope of this document.

Keymaps:

ja_JP.IBM-eucJP.imkeymap

Ja_JP.IBM-932.imkeymap

Ja_JP.IBM-943.imkeymap

Keysyms:

The JIM uses the keysyms in the **XK_KATAKANA**, **XK_LATIN1**, and **XK_MISCELLANY** groups.

Reserved Keysyms:

XK_BunsetsuYomi	0x1800ff05	Back a phrase to Yomi
XK_MaeKouho	0x1800ff04	Previous candidate
XK_ZenKouho	0x1800ff01	All candidates.
XK_KanjiBangou	0x1800ff02	Kanji number input.
XK_HenkanMenu	0x1800ff03	Changes conversion mode.
XK_LeftDouble	0x1800ff06	Moves cursor two characters left.
XK_RightDouble	0x1800ff07	Moves cursor two characters right.
XK_LeftPhrase	0x1800ff08	Reserved for future use.
XK_RightPhrase	0x1800ff09	Reserved for future use.
XK_ErInput	0x1800ff0a	Discards the current pre-edit string
XK_Resetreset	0x1800ff0b	Reset

The preceding keysyms are unique to the input method of this system.

XK_Kanji Convert Hiragana to Kanji.
XK_Muhenkan Cancels conversion.
XK_Romaji Puts JIM in Romaji input mode.
XK_Hiragana Puts JIM in Hiragana input mode.
XK_Katakana Puts JIM in Katakana input mode.
XK_Zenkaku_Hankaku Toggles between full-width and half-width character input mode.
XK_Touroku Registers a word to the user dictionary.
XK_Eisu_toggle Puts JIM in alphanumeric input mode.

- Modifiers:

ShiftMask 0x01
LockMask 0x02
ControlMask 0x04
Mod1Mask (Left-Alt) 0x08
Mod2Mask (Right-Alt) 0x10

- Internal Modifiers:

Kana 0x20
Romaji 0x40

Korean Input Method (KIM)

The Korean EUC code set consists of two main character groups:

- ASCII (English)
- Hangul (Korean characters)

The Hangul code set includes Hangul and Hanja (Chinese) characters. One Hangul character can comprise several consonants and vowels. However, most Hangul words can be expressed in Hanja. Each Hanja character has its own meaning and is thus more specific than Hangul.

The current Korean standard code set, KSC5601, contains 8224 Hangul, Hanja, and special characters. To comply with the Korean standard Extended UNIX Code (EUC), this code set is assigned to CS1 of the EUC.

Input of characters can be accomplished through:

- ASCII

  ASCII mode is used for entering English characters.
- Hangul

  The XK_Hangul key invokes Hangul mode, which must be used to enter Hangul characters. Once Hangul mode is invoked, the KIM composes incoming consonants and vowels according to Hangul composition rules. A Hangul character is composed of a consonant followed by a vowel. A final consonant is optional. If incoming characters violate the construct rule, a warning beep is sounded.

  There are about 1500 special characters in the standard code set. These characters must be entered with the Code Input function of the KIM. The Code Input key invokes the Code Input function. When the Code Input function is invoked, the code point for a desired character can be entered in the Code Input auxiliary window.
- Hanja

  The XK_Hangul_Hanja key invokes the Hanja mode. Hanja characters can only be converted from the appropriate Hangul character. There are two modes for Hangul-to-Hanja Conversion (HHC): single-candidate and multi-candidate. In this context, a candidate is a selection of possible character choices.

  In single-candidate mode, the candidates are displayed one by one on the command line. In multi-candidate mode, up to ten candidates at a time are displayed in an auxiliary window.

  When the Hanja conversion mode is employed, any Hangul character can be converted into Hanja when the Conversion key is pressed. Similarly, any Hanja word can be converted to the appropriate Hangul word.

  Hanja can also be entered with the Code Input function in the same manner used for entering Hangul.

To allow for these conversions, the following special keys appear on the 106-key Korean keyboard.

Special Korean Keys		
Key Function	**Keysym**	**Description of Function**
Hangul/English toggle key	XK_Hangul	Toggles between Hangul and English modes
Hanja toggle key	XK_Hangul_Hanja	Toggles Hanja mode on and off
Code Input key	XK_Hangul_ Codeinput	Invokes the Code Input function, which allows characters to be entered by their code points
HHC All-Candidate key	XK_Hangul_ MultipleCandidate	Invokes the multi-candidate mode
HHC Conversion key	XK_Hangul_ Conversion	Invokes the single-candidate mode and also scrolls forward through the candidates in both single-candidate and multi-candidate modes
HHC Non-Conversion key	XK_Hangul_ NonConversion	Scrolls backwards through the candidates

Latvian Input Method (LVIM)

The Latvian Input Method (LVIM) is similar to the Single-Byte Input Method (SIM), except that it is customized for processing the Latvian keyboard. The features of LVIM are:

- Supports QWERTY and Ergonomic groups, as two main groups. There are two more supplementary groups which are accessible through dead keys from both main groups:
  - Pressing the left-alt key and left-shift key simultaneously, puts keyboard in the Ergonomic group.
  - Pressing the left-alt key and right-shift key simultaneously, puts keyboard in the QWERTY group.
- Supports the IBM-921 code set.

Keymap:

Lv_LV.IBM-921.imkeymap

Lithuanian Input Method (LTIM)

The Lithuanian Input Method (LTIM) is similar to the Single-Byte Input Method (SIM), except that it is customized for processing the Lithuanian keyboard. The features of LTIM are:

- Supports Programmed and Lithuanian groups, as two main groups. There are two more supplementary groups which are accessible through dead keys from both main groups.
  - Pressing the left-alt key and left-shift key simultaneously, puts keyboard in the Lithuanian group.
  - Pressing the left-alt key and right-shift key simultaneously, puts keyboard in the Programmed group.
- Supports the IBM-921 code set.

Keymap:

Lt_LT.IBM-921.imkeymap

Thai Input Method (THIM)

The Thai Input Method is similar to the Single-Byte Input Method (SIM), except that it is customized for processing the Thai language.

Specifically, it is designed to prevent entry of combinations of Thai characters (consonants, upper/lower vowels, tone marks) that are invalid in the Thai language. The features of the THIM are:

- Supports Latin and Thai groups, as the two main groups on the keyboard.
  - Pressing the left-alt key and left-shift key puts the keyboard in the Thai group.
  - Pressing the left-alt key and right-shift key puts the keyboard in the Latin group.
- Supports the TIS-620 codeset.

Keymap:

th_TH.TIS-620.imkeymap

Vietnamese Input Method (VNIM)

The Vienamese Input Method is similar to the Single-Byte Input Method (SIM), except that it is customized for processing the Vietnamese language.

Specifically, it is designed to prevent entry of combinations of Vietnamese characters (tone marks), that are invalid in the Vietnamese language. The Vietnamese tone mark characters can only be entered immediately after one of the Vietnamese vowels (a, e, i, o, u, y, a-circumflex, e-circumflex, o-circumflex, a-breve, o-horn, or u-horn).

The Vietnamese Input method supports a single keyboard layer, including some pre-composed characters and Vietnamese tone marks.

- Supports the IBM-1129 codeset.

Keymap:

Vi_VN.IBM-1129.imkeymap

Simplified Chinese Input Method (ZIM)

The IBM-eucCN code set consists of two character groups:

- ASCII (English)
- Simplified Chinese (GB2312.1980)

Simplified Chinese (GB2312.1980) contains 6,763 Chinese characters. It is divided into two parts: first level and second level. The characters belonging to the first level are frequently used. Each character is comprised of one to six components known as radicals.

The pronunciation of Simplified Chinese is represented by phonetic symbols called Bo-Po-Mo-Fo. There are 25 phonetic symbols. A Simplified Chinese character is represented by one to three phonetic symbols.

ZIM features the following characteristics:

- Three commonly used input methods:

  PinYin (including legend). An input method based on phonetic combinations.

  English-to-Chinese. An input method based on word-by-word translation from English to Chinese.

  ABC. An intelligent input method based on phonetic combinations.
- Half-width and full-width character input. Supports ASCII characters in both single-byte and multibyte modes.
- Auxiliary pop-up window to support all the candidate lists. PinYin, English-to-Chinese, and ABC generate a list of possible characters that contain the same sound symbols or radicals. Users select the desired characters by pressing the conversion key.
- Over-the-spot pre-editing drawing area. Allows entry of radicals in reverse video area that temporarily covers the text line. The complete character is sent to the editor by pressing the conversion key.

The ZIM files are in the **/usr/lib/nls/loc** directory.

The ZIM keymap is in the **/usr/lib/nls/loc/zh_CN.IBM-eucCN.imkeymap** directory.

Simplified Chinese Character Processing

ZIM is invoked by pressing one of the input method keys. Each radical or phonetic symbol is assigned to a key. The user inputs radicals or phonetic symbols to an over-the-spot pre-editing area. PinYin, English-to-Chinese, and ABC input generate a list of candidates that appear in a pop-up window. The user chooses the desired character by selecting the candidate number. Invalid input generates a beep and an error message.

The following special keys for the Simplified Chinese input method are defined on the Simplified Chinese 101-key keyboard.

Special Simplified Chinese Keys		
Key Function	**Keysym**	**Description of Function**
Phonetic Shift key	XK_Pin_Yin	Invokes the phonetic input method.
Legend Shift key	XK_Legend	Under phonetic input method, invokes phonetic legend mode.
English-to-Chinese Shift key	XK_English _Chinese	Invokes the English-to-Chinese input method.
ABC Shift key	XK_ABC	Invokes ABC input method.
ABC Set Option Shift key	XK_ABC_Set _Option	Under ABC input method, invokes ABC Set Option mode.
Half/Full-Width toggle Shift key	XK_Half_Full	Toggles between half-width and full-width.
Conversion Shift key	XK_Convert	Converts radical and phonetic symbols into characters. Displays the candidate list in an auxiliary window, if needed.
Non-Conversion Shift key	XK_Non _Convert	Interprets a phonetic symbol as a character.
English/Numeric key	XK_Alph_Num	Invokes ASCII mode.
Five Stroke Shift key	XK_Five_Stroke	Invokes five stroke input method.
User Defined Shift key	XK_User _Defined	Invokes user-defined input method.

Simplified Chinese Input Method (ZIM-UCS)

The UCS-2 code set consists of almost all character groups. For the ZH_CN locale, there are three character groups:

- ASCII (English)
- Glyphs
- Chinese, Japanese, and Korean (CJK) Characters (unification characters)

The CJK character set contains 20,992 character positions, but only 20,902 positions are assigned to Chinese characters.

The pronunciation of Chinese is represented by phonetic symbols called Bo-Po-Mo-Fo. There are 25 phonetic symbols. Chinese characters are represented by one to three phonetic symbols.

UCS-ZIM features the following characteristics:

- Four commonly used input methods:

  Tsang-Jye. An input method based on the construction of Chinese characters.

  PinYin (including legend). An input method based on phonetic combinations.

  English-to-Chinese. An input method based on word-by-word translation from English to Chinese.

  ABC (Chinese Word Conversion). An input method based on phonetic combinations and Chinese words.
- Half-width and full-width character input. Supports ASCII characters in both single-byte and multibyte modes.

- Auxiliary pop-up window to support all the candidate lists. PinYin, English-to-Chinese, and ABC generate a list of possible characters that contain the same sound symbols or radicals. Users select the desired characters by pressing the conversion key.
- Over-the-spot pre-editing drawing area. Allows entry of radicals in reverse video area that temporarily covers the text line. The complete character is sent to the editor by pressing the conversion key.

The UCS-ZIM files are in the **/usr/lib/nls/loc** directory.

The UCS-ZIM keymap is in the **/usr/lib/nls/loc/ZH_CN.UTF-8.imkeymap** directory.

Chinese (CJK) Character Processing

UCS-ZIM is invoked by pressing one of the input method keys. Each radical or phonetic symbol is assigned to a key. The user inputs radicals or phonetic symbols to an over-the-spot pre-editing area. For Tsang-Jye and five stroke input, a character is generated when the conversion key is pressed. PinYin, English-to-Chinese, and ABC input generate a list of candidates that appear in a pop-up window. The user chooses the desired character by selecting the candidate number. Invalid input generates a beep and an error message. The glyphs can be input using the ABC input method.

The following special keys for the UCS-Chinese input method are defined on the Simplified Chinese 101-key keyboard.

Special UCS-Chinese Keys		
Key Function	**Keysym**	**Description of Function**
Tsang-Jye Shift key	XK_Tsang_Jye	Invokes Tsang-Jye input method.
Phonetic Shift key	XK_Pin_Yin	Invokes the phonetic input method.
Legend Shift key	XK_Legend	Under phonetic input method, invokes phonetic legend mode.
English-to-Chinese Shift key	XK_English _Chinese	Invokes the English-to-Chinese input method.
ABC Shift key	XK_ABC	Invokes ABC input method.
Five Stroke Shift key	XK_Five_Stroke	Invokes Five Stroke input method.
IM Set Option key	XK_IMED_Set _option	Under phonetic and ABC input method, invokes territory set mode.
Half/Full-Width Toggle Shift key	XK_Half_Full	Toggles between half-width and full-width.
Conversion Shift key	XK_Convert	Converts radical and phonetic symbols into characters. Displays the candidates, if needed.
Non-Conversion Shift key	XK_Non _Convert	Interprets a phonetic symbol as a character.
English/Numeric key	XK_Alph_Num	Invokes ASCII mode.
User-Defined Shift key	XK_User _Defined	Invokes User defined input method.

Single-Byte Input Method

The Single-Byte Input Method (SIM) is the standard that supports most of the locales. It is a mapping function that supports simple composing defined on workstation keyboards associated with single-byte locales.

SIM supports any keyboard, code set, and language that the **keycomp** command can describe. You can customize SIM using imkeymaps. The coded strings returned by the input method depend on the imkeymap.

Most single-byte locales share one SIM. The SIM features are:

- Supports 101-key and 102-key keyboard mapping.
- Supports Alt-Numpad composing.

  When you press the Alt key, the input method composes a character by using the next three numeric keys pressed. The three numeric keys represent the decimal encoding of the character. For example, entering the sequence `XK_0, XK_9, XK_7` maps to the character *a* (097).
- Supports the Num-Lock state for the numeric keypad.
- Supports diacritical composing.

  The e-umlaut key is an example of diacritical composing. To compose e-umlaut, the user presses the appropriate diacritical key (umlaut) followed by an alphabetic key (e). The specific set of diacritical keys in use depend on the locale and keyboard definition. When a space follows a diacritical key, the diacritical character represented by the key is returned if it is in the locale's code set.
- Does not require callback functions.
- Located in the **/usr/lib/nls/loc/sbcs.im** file. Most of the other localized input methods are aliases to this file.
- Keymaps:

cs_CZ.ISO8859-2.imkeymap	
da_DK.ISO8859-1.imkeymap	Da_DK.IBM-850.imkeymap
de_CH.ISO8859-1.imkeymap	De_CH.IBM-850.imkeymap
de_DE.ISO8859-1.imkeymap	De_DE.IBM-850.imkeymap
en_GB.ISO8859-1.imkeymap	En_GB.IBM-850.imkeymap
en_GB.ISO8859-1@alt.imkeymap	En_GB.IBM-850@alt.imkeymap
en_US.ISO8859-1.imkeymap	En_US.IBM-850.imkeymap
es_ES.ISO8859-1.imkeymap	Es_ES.IBM-850.imkeymap
Et_EE.IBM-922 - imkeymap	
pl_PL.ISO8859-2@alt.imkeymap	
sq_AL.ISO8859-1.imkeymap	
fi_FI.ISO8859-1.imkeymap	Fi_FI.IBM-850.imkeymap
fi_FI.ISO8859-1@alt.imkeymap	Fi_FI.IBM-850@alt.imkeymap
fr_BE.ISO8859-1.imkeymap	Fr_BE.IBM-850.imkeymap
fr_CA.ISO8859-1.imkeymap	Fr_CA.IBM-850.imkeymap
fr_CH.ISO8859-1.imkeymap	Fr_CH.IBM-850.imkeymap
fr_FR.ISO8859-1.imkeymap	Fr_FR.IBM-850.imkeymap
fr_FR.ISO8859-1@alt.imkeymap	Fr_FR.IBM-850@alt.imkeymap
hr_HR.ISO8859-2.imkeymap	
hu_HU.ISO8859-2.imkeymap	
is_IS.ISO8859-1.imkeymap	Is_IS.IBM-850.imkeymap
it_IT.ISO8859-1.imkeymap	It_IT.IBM-850.imkeymap
it_IT.ISO8859-1@alt.imkeymap	It_IT.IBM-850@alt.imkeymap
nl_BE.ISO8859-1.imkeymap	Nl_BE.IBM-850.imkeymap
nl_NL.ISO8859-1.imkeymap	Nl_NL.IBM-850.imkeymap
no_NO.ISO8859-1.imkeymap	No_NO.IBM-850.imkeymap
pl_PL.ISO8859-2.imkeymap	
pt_BR.ISO8859-1.imkeymap	
pt_PT.ISO8859-1.imkeymap	Pt_PT.IBM-850.imkeymap
ro_RO.ISO8859-2.imkeymap	
sh_SP.ISO8859-2.imkeymap	
sl_SI.ISO8859-2.imkeymap	
sk_SK.ISO8859-2.imkeymap	

sv_SE.ISO8859-1.imkeymap
sv_SE.ISO8859-1@alt.imkeymap
tr_TR.ISO8859-1.imkeymap
Sv_SE.IBM-850.imkeymap
Sv_SE.IBM-850@alt.imkeymap

- Reserved keysyms:

XK_dead_acute	0x180000b4
XK_dead_grave	0x18000060
XK_dead_circumflex	0x1800005e
XK_dead_diaeresis	0x180000a8
XK_dead_tilde	0x1800007e
XK_dead_caron	0x180001b7
XK_dead_breve	0x180001a2
XK_dead_doubleacute	0x180001bd
XK_dead_degree	0x180000b0
XK_dead_abovedot	0x180001ff
XK_dead_macron	0x180000af
XK_dead_cedilla	0x180000b8
XK_dead_ogonek	0x180001b2
XK_dead_accentdieresis	0x180007ae

The preceding keysyms are unique to this input method and are described in the **/usr/include/X11/aix_keysym.h** file.

- Modifiers:

ShiftMask	0x01
LockMask	0x02
ControlMask	0x04
Mod1Mask (Left-Alt)	0x08
Mod2Mask (Right-Alt)	0x10
Mod5Mask (Num Lock)	0x80

Traditional Chinese Input Method (TIM)

The Traditional Chinese code sets consist of two character groups:

- ASCII (English)
- Traditional Chinese characters

The Traditional Chinese character set contains more than 100,000 characters, but only about 5000 are frequently used. Each character is comprised of one to five components known as radicals.

The pronunciation of Traditional Chinese is represented by phonetic symbols called Dsu-Yin or Bo-Po-Mo-Fo. There are 37 phonetic symbols plus four intonation indicators. Chinese characters are represented by one to three phonetic symbols. The character can include one intonation symbol. The omission of an intonation symbol implies a fifth intonation accent.

TIM Features

TIM features the following characteristics:

- Five commonly used input methods:
  - Tsang-Jye. Supports radicals to generate a character. Most frequently used by data entry personnel.
  - Simplified Tsang-Jye. Supports wildcard input and radicals. This input method also allows entry of partial characters.

  - Phonetic symbols. Inputs a character based on its pronunciation.
  - Internal Code. Generates characters by EUC hexadecimal, code point input.
  - Decimal value. Generates characters by decimal value. Can be invoked from any of the input modes.
- Half-width and full-width character input. Supports ASCII characters in both single-byte and multibyte modes.
- System-defined and user-definable character input.
- Auxiliary pop-up window to support all the candidate lists. Simplified Tsang-Jye and phonetic input methods generate a list of character candidates that contains the same input radicals or sound symbols. Users pick the desired characters by pressing the corresponding number.
- Over-the-spot pre-editing drawing area. Allows entry of radicals in reverse video area that temporarily covers the text line. The complete character is sent to the editor by pressing the conversion key.

The TIM file is found in the **/usr/lib/nls/loc/TW.im** directory.

The TIM keymap is found in the **/usr/lib/nls/loc/zh_TW.IBM-eucTW.imkeymap** directory.

Traditional Chinese Character Processing

TIM is invoked by pressing one of the input-method keys. Each radical or phonetic symbol is assigned to a key. The user inputs radicals or phonetic symbols to an over-the-spot pre-editing area. For Tsang-Jye and Internal Code input, a character is generated when the conversion key is pressed. Simplified Tsang-Jye and Phonetic input generate a list of candidates that appear in a pop-up window. The user chooses the desired character by selecting the candidate number. Invalid input generates a beep and an error message.

The following special keys for the Traditional Chinese Input Method are defined on the Traditional Chinese 106-key keyboard.

Special Traditional Chinese Keys		
Key Function	**Keysym**	**Description of Function**
Tsang-Jye Shift key	XK_Chinese _Tsangjei	Invokes both the Tsang-Jye and Simplified Tsang-Jye input methods.
Phonetic Shift key	XK_Chinese _Phonetic	Invokes the Phonetic input method.
Half/Full-Width toggle key	XK_Chinese _Full_Half	Toggles between half-width and full-width.
Conversion key	XK_Convert	Converts radical and phonetic symbols or EUC code symbols into characters. Displays the candidate list in an auxiliary window, if needed.
Non-Conversion key	XK_Non _Convert	Interprets a phonetic symbol as a character.
English/Numeric key	XK_Alph_Num	Invokes ASCII mode.
ALT-Tsang-Jye Shift key	XK_Internal _Code	Invokes Internal Code input method.
ALT plus number keypad		Invoke the decimal value input method.

Universal Input Method

The Universal Input Method is used in the Unicode/UTF-8 locales to provide complete multlingual input method support. Features of the Universal Input Method are:

- Supports Input Method Switching

  - Pressing the `Ctrl` key and the left `Alt` and the letter `i` simultaneously, presents a menu listing the other available input methods. Selecting an input method from the list remaps the keyboard and loads the given input method, allowing character entry using the loaded input method.
- Supports Point and Click Character Input
  - Pressing the `Ctrl` key and the left `Alt` and the letter `l` simultaneously, presents a menu listing the various categories of characters contained in the Unicode standard. Selecting a character list presents a matrix of the available characters from the list. Clicking on a given character will then send that character through the input method to the application.
  - Pressing the `Ctrl` key and the left `Alt` and the letter `c` returns to the application, or if already in the application, returns to the most recently used character list for point and click character entry.
- Supports the UTF-8 code set.

Keymap:

XX_XX.UTF-8.imkeymap

List of Reserved Keysyms

The keysyms listed are reserved for use by the input methods:

XK_dead_acute	0x180000b4
XK_dead_grave	0x18000060
XK_dead_circumflex	0x1800005e
XK_dead_diaeresis	0x180000a8
XK_dead_tilde	0x1800007e
XK_dead_caron	0x180001b7
XK_dead_breve	0x180001a2
XK_dead_doubleacute	0x180001bd
XK_dead_degree	0x180000b0
XK_dead_abovedot	0x180001ff
XK_dead_macron	0x180000af
XK_dead_cedilla	0x180000b8
XK_dead_ogonek	0x180001b2
XK_dead_accentdieresis	0x180007ae
XK_BunsetsuYomi	0x1800ff05
XK_MaeKouho	0x1800ff04
XK_ZenKouho	0x1800ff01
XK_KanjiBangou	0x1800ff02
XK_HenkanMenu	0x1800ff03
XK_LeftDouble	0x1800ff06
XK_RightDouble	0x1800ff07
XK_LeftPhrase	0x1800ff08
XK_RightPhrase	0x1800ff09
XK_ErInput	0x1800ff0a
XK_Reset	0x1800ff0b

Reserved Keysyms for Traditional Chinese

XK_Full_Size	0xff42
XK_Phonetic	0xff48
XK_Alph_Num	0xaff50
XK_Non_Convert	0xaff52
XK_Convert	0xaff51

XK_Tsang_Jye	0xff47
XK_Internal_Code	0xff4a

Reserved Keysyms for Simplified Chinese (ZIM and ZIM-UCS)

XK_Alph_Num	0xaff47
XK_Non_Convert	0xaff59
XK_Row_Column	0xaff48
XK_PinYin	0xaff49
XK_English_Chinese	0xaff50
XK_ABC	0xaff51
XK_Fivestroke	0xaff62
XK_User-defined	0xaff56
XK_Legend	0xaff55
XK_ABC_Set_Option	0xaff60
XK_Half_full	0xaff53

Message Facility Overview for Programming

To facilitate translations of messages into various languages and make them available to a program based on a user's locale, it is necessary to keep messages separate from the program by providing them in the form of message catalogs that the program can access at run time. To aid in this task, commands and subroutines are provided by the Message Facility.

Message source files containing application messages are created by the programmer and converted to message catalogs. The application uses these catalogs to retrieve and display messages, as needed. Translating message source files into other languages and then converting the files to message catalogs does not require changing and recompiling a program.

The following information is provided for understanding the Message Facility:

- "Creating a Message Source File"
- "Creating a Message Catalog" on page 484
- "Displaying Messages outside of an Application Program" on page 486

Creating a Message Source File

The Message Facility provides commands and subroutines to retrieve and display program messages located in externalized message catalogs. A programmer creates a message source file containing application messages and converts it to a message catalog with the **gencat** command.

To create a message-text source file, open a file using any text editor. Enter a message identification number or symbolic identifier. Then enter the message text as shown in the following example:

```
1 message-text   $       (This message is numbered)
2 message-text   $       (This message is numbered)
OUTMSG message-text  $   (This message has a symbolic identifier \
                          called OUTMSG)
4 message-text   $       (This message is numbered)
```

Usage Considerations

Consider the following:

- One blank character must exist between the message ID number or identifier and the message text.

- A symbolic identifier must begin with an alphabetical character and can contain only letters of the alphabet, decimal digits, and underscores.
- The first character of a symbolic identifier cannot be a digit.
- The maximum length of a symbolic identifier is 64 bytes.
- Message ID numbers must be assigned in ascending order within a single message set, but need not be contiguous. 0 (zero) is not a valid message ID number.
- Message ID numbers must be assigned as if intervening symbolic identifiers are also numbered. If, for example, you had numbered the lines as in the previous example, `1`, `2`, `OUTMSG`, and `3`, the program would contain an error. This is because the **mkcatdefs** command also assigns numbers to symbolic identifiers, and would have assigned number `3` to the `OUTMSG` symbolic identifier.

  Note: Symbolic identifiers are specific to the Message Facility. Portability of message source files can be affected by the use of symbolic identifiers.

Adding Comments to the Message Source File

You can include a comment anywhere in a message source file except within message text. Leave at least one space or tab (blank) after the $ (dollar sign). The following is an example of a comment:

```
$ This is a comment.
```

Comments do not appear in the message catalog generated from the message source file.

Comments can help developers in the process of maintaining message source files, translators in the process of translation, and writers in the process of editing and documenting messages. Use comments to identify what variables, such as `%`**s**, `%`**c**, and `%`**d,** represent. For example, create a note that states whether the variable refers to a user, file, directory, or flag. Comments also should be used to identify obsolete messages.

For clarity, you should place a comment line directly beneath the message to which it refers, rather than at the bottom of the message catalog. Global comments for an entire set can be placed directly below the **$set** directive.

Continuing Messages on the Next Line

All text following the blank after the message number is included as message text, up to the end of the line. Use the escape character \ (backslash) to continue message text on the following line. The \ (backslash) must be the last character on the line as in the following example:

```
5 This is the text associated with \
message number 5.
```

These two physical lines define the single-line message:

```
This is the text associated with message number 5.
```

Note: The use of more than one blank character after the message number or symbolic identifier is specific to the Message Facility. Portability of message source files can be affected by the use of more than one blank.

Including Special Characters in the Message Text

The \ (backslash) can be used to insert special characters into the message text. These special characters are:

\n Inserts a new-line character.
\t Inserts a horizontal tab character.
\v Inserts a vertical tab character.

\b Inserts a backspace character.

\r Inserts a carriage-return character.

\f Inserts a form-feed character.

**** Inserts a \ (backslash) character.

*****ddd* Inserts a single-byte character associated with the octal value represented by the valid octal digits *ddd*.

> **Note:** One, two, or three octal digits can be specified. However, you must include a leading zero if the characters following the octal digits are also valid octal digits. For example, the octal value for $ (dollar sign) is 44. To display $5.00, use \0445.00, not \445.00, or the 5 will be parsed as part of the octal value.

\x*dd* Inserts a single-byte character associated with the hexadecimal value represented by the two valid hexadecimal digits *dd*. You must include a leading zero to avoid parsing errors (see the note about *****ddd*).

\x*dddd* Inserts a double-byte character associated with the hexadecimal value represented by the four valid hexadecimal digits *dddd*. You must include a leading zero to avoid parsing errors (see the note about *****ddd*).

Defining a Character to Delimit Message Text

You can use the **$quote** directive in a message source file to define a character for delimiting message text. This character should be an ASCII character. The format is:

```
$quote [character] [comment]
```

Use the specified character before and after the message text. In the following example, the **$quote** directive sets the quote character to _ (underscore), and then disables it before the last message, which contains quotation marks:

```
$quote _   Use an underscore to delimit message text
$set MSFAC         Message Facility - symbolic identifiers
SYM_FORM   _Symbolic identifiers can contain alphanumeric \
characters or the \_ (underscore character)\n_
SYM_LEN    _Symbolic identifiers can be up to 65 \
characters long \n_
5              _You can mix symbolic identifiers and numbers \n_
$quote
MSG_H     Remember to include the _msg_h_ file in your program\n
```

The last **$quote** directive in the previous example disables the underscore character.

In the following example, the **$quote** directive defines ″ (double quotation marks) as the quote character. The quote character must be the first non-blank character following the message number. Any text following the next occurrence of the quote character is ignored.

```
$quote "   Use a double quote to delimit message text
$set 10            Message Facility - Quote command messages
1   "Use the $quote directive to define a character \
\n for delimiting message text"
2   "You can include the \"quote\" character in a message \n \
by placing a \\ in front of it"
3   You can include the "quote" character in a message \n \
by having another character as the first nonblank \
\n character after the message ID number
$quote
4   You can disable the quote mechanism by \n \
using the $quote directive without a character \n\
after it
```

This example illustrates two ways the quote character can be included in message text:

- Place a \ (backslash) in front of the quote character.
- Use some other character as the first non-blank character following the message number. This disables the quote character only for that message.

The example also shows the following:

- A \ (backslash) is still required to split a quoted message across lines.
- To display a \ (backslash) in a message, place another \ (backslash) in front of it.
- You can format a message with a new-line character by using **\n**.
- Using the **$quote** directive with no character argument disables the quote mechanism.

Assigning Message Set Numbers and Message ID Numbers

All message sets require a set number or symbolic identifier. Use the **$set** directive in a source file to give a group of messages a number or identifier:

$set *n* [*comment*]

The message set number is specified by the value of *n*, a number between 1 and **NL_SETMAX**. Instead of a number, you can use a symbolic identifier. All messages following the **$set** directive are assigned to that set number until the next occurrence of a **$set** directive. The default set number is 1. Set numbers must be assigned in ascending order, but need not be in series. Empty sets are created for skipped numbers. However, large gaps in the number sequence decrease efficiency and performance. Moreover, performance is not enhanced by using more than one set number in a catalog.

You can also include a comment in the **$set** directive, as follows:

```
$set 10  Communication Error Messages
$set OUTMSGS  Output Error Messages
```

Many AIX message sets have a symbolic identifier of the form **MS**_*PROG*, where **MS** represents Message Set and *PROG* is the name of the program or utility related to the message set. For example:

```
$set MS_WC  Message Set for the wc Utility
$set MS_XLC1  Message Set 1 for the C For AIX compiler
$set MS_XLC2  Message Set 2 for the C For AIX compiler
```

Removing Messages from a Catalog

The **$delset** directive removes all of the messages belonging to a specified set from an existing catalog:

$delset *n* [*comment*]

The message set is specified by *n*. The **$delset** directive must be placed in the proper set-number order with respect to any **$set** directives in the same source file. You can also include a comment in the **$delset** directive.

Length of Message Text

The **$len** directive establishes the maximum display length of message text:

$len [*n* [*comment*]]

If *n* is not specified or if the **$len** directive is not included, the message text display is set to the **NL_TEXTMAX** value. The message-text display length is the maximum number of bytes allowed for a message. Any subsequent specification of a **$len** directive overrides a previous specification. The value of *n* cannot exceed the **NL_TEXTMAX** value.

Content of Message Text

Cause and Recovery Information: Whenever possible, tell users exactly what has happened and what they can do to remedy the situation. The following example shows how cause and recovery information can improve a message:

```
Original  Message:   Bad arg
```

```
Revised Message:   Specify year as a value between 1 and 9999.
```

The message `Bad arg` does not help users much; whereas the message `Do not specify more than 2 files on the command line` tells users exactly what they must do to make the command work. Similarly, the message `Line too long` does not give users recovery information. The message `Line cannot exceed 20 characters` provides the missing information.

Examples of Message Source Files

1. The following example message source file uses numbers for message ID numbers and for message set numbers:

```
$ This is a message source file sample.
$ Define the Quote Character.
$quote "
$set 1 This is the set 1 of messages.
1 "The specified file does not have read permission on\n"
2 "The %1$s file and the %2$s file are same\n"
3 "Hello world!\n"
$Define the quote character
$quote '
$set 2 This is the set 2 of messages
1       'fieldef: Cannot open %1$s \n'
2       'Hello world\n'
```

2. The following example message source file uses symbolic identifiers for message ID numbers and for message set numbers:

```
$ This is a message source file sample.
$ Define the Quote Character.
$quote "
$set MS_SET1 This is the set 1 of messages.
MSG_1   "The specified file does not have read permission on\n"
MSG_2   "The %1$s file and the %2$s file are same\n"
MSG_3   "Hello world\n"
$Define the quote character
$quote
$set 2 This is the set 2 of messages.
$EMSG_1 'fieldef: Cannot open %1$s\n'
$EMSG_2 'Hello world!\n'
```

3. The following examples show how symbolic identifiers can make the specification of a message more understandable:

```
catgets(cd, 1, 1, "default message")
```

```
catgets(cd, MS_SET1, MSG_1, "default message")
```

Creating a Message Catalog

The Message Facility provides commands and subroutines to retrieve and display program messages located in externalized message catalogs. A programmer creates a message source file containing application messages and converts it to a message catalog. Translating message source files into other languages and then converting the files to message catalogs does not require changing or recompiling a program.

To create a message catalog, process your completed message source file with the message facility's **gencat** command. This command can be used three ways:

- Use the **gencat** command to process a message source file containing set numbers, message ID numbers, and message text. Message source files containing symbolic identifiers cannot be processed directly by the **gencat** command. The following example uses the information in the `x.msg` message source file to generate a catalog file:

```
gencat x.cat x.msg
```

- Use the **mkcatdefs** command to preprocess a message source file containing symbolic identifiers. The resulting file is then piped to the **gencat** command. The **mkcatdefs** command produces a

*SymbolName*_**msg.h** file containing definition statements. These statements equate symbolic identifiers with set numbers and message ID numbers assigned by the **mkcatdefs** command. The *SymbolName*_**msg.h** file should be included in programs using these symbolic identifiers. The **mkcatdefs** command is specific to AIX. The following example uses the information in the `x.msg` message source file to generate the `x_msg.h` header file:

```
mkcatdefs x x.msg
```

- Use the **runcat** command to automatically process a source file containing symbolic identifiers. The **runcat** command invokes the **mkcatdefs** command and pipes its output to the **gencat** command. The **runcat** command is specific to AIX. The following example uses the information in the `x.msg` message source file to generate the `x_msg.h` header file and the `X.cat` catalog file:

  ```
  runcat x x.msg
  ```

The preceding example is equivalent to the following example:

```
mkcatdefs x x.msg | gencat x.cat
```

If a message catalog with the name specified by the *CatalogFile* parameter exists, the **gencat** command modifies the catalog according to the statements in the message source files. If a message catalog does not exist, the **gencat** command creates a catalog file with the name specified by the *CatalogFile* parameter.

You can specify any number of message text source files. Multiple files are processed in the sequence you specify. Each successive source file modifies the catalog. If you do not specify a source file, the **gencat** command accepts message source data from standard input.

Catalog Sizing

A message catalog can be virtually any size. The maximum numbers of sets in a catalog, messages in a catalog, and bytes in a message are defined in the **/usr/include/limits.h** file by the following macros:

NL_SETMAX	Specifies the maximum number of set numbers that can be specified by the **$set** directive. If the **NL_SETMAX** limit is exceeded, the **gencat** command issues an error message and does not create or update the message catalog.
NL_MSGMAX	Specifies the maximum number of message ID numbers allowed by the system. If the **NL_MSGMAX** limit is exceeded, the **gencat** command issues an error message and does not create or update the message catalog.
NL_TEXTMAX	Specifies the maximum number of bytes a message can contain. If the **NL_TEXTMAX** limit is exceeded, the **gencat** command issues an error message and does not create or update the message catalog.

Examples

1. This example shows how to create a message catalog from a source file containing message identification numbers. The following is the text of the `hello.msg` message source file:

   ```
   $ file: hello.msg
   $set 1    prompts
   1 Please, enter your name.
   2 Hello, %s \n
   $ end of file: hello.msg
   ```

   To create the `hello.cat` message catalog from the `hello.msg` source file, enter:

   ```
   gencat hello.cat hello.msg
   ```

2. This example shows how to create a message catalog from a source file with symbolic references. The following is the text of the `hello.msg` message source file that contains symbolic references to the message set and the messages:

```
$ file: hello.msg
$quote "
$set PROMPTS
PLEASE  "Please, enter your name."
HELLO   "Hello, %s \n"
$ end of file: hello.msg
```

The following is the text of the `msgerrs.msg` message source file that contains error messages that can be referenced by their symbolic IDs:

```
$ file: msgerrs.msg
$quote "
$set CAT_ERRORS
MAXOPEN  "Cannot open message catalog %s \n \
Maximum number of catalogs already open "
NOT_EX "File %s not executable \n "
$set MSG_ERRORS
NOT_FOUND "Message %1$d, Set %2$d not found \n "
$ end of file: msgerrs.msg
```

To process the `hello.msg` and `msgerrs` message source files, enter:

```
runcat hello hello.msg
runcat msgerrs msgerrs.msg /usr/lib/nls/msg/$LANG/msgerrs.cat
```

The **runcat** command invokes the **mkcatdefs** and **gencat** commands. The first call to the **runcat** command takes the `hello.msg` source file and uses the second parameter, `hello`, to produce the `hello.cat` message catalog and the `hello_msg.h` definition file.

The `hello_msg.h` definition file contains symbolic names for the message catalog and symbolic IDs for the messages and sets. The symbolic name for the `hello.cat` message catalog is `MF_HELLO`. This name is produced automatically by the **mkcatdefs** command.

The second call to the **runcat** command takes the `msgerrs.msg` source file and uses the first parameter, `msgerrs`, to produce the `msgerrs_msg.h` definition file.

Since the third parameter, `/usr/lib/nls/msg/$LANG/msgerrs.cat`, is present, the **runcat** command uses this parameter for the catalog file name. This parameter is an absolute path name that specifies exactly where the **runcat** command must put the file. The symbolic name for the `msgerrs.cat` catalog is `MF_MSGERRS`.

Displaying Messages outside of an Application Program

The following commands allow you to display messages outside of an application program. These commands are specific to AIX.

dspcat Displays the messages contained in the specified message catalog. The following example displays the messages located in the `x.cat` message source file:

```
dspcat x.cat
```

dspmsg Displays a single message from a message catalog. The following example displays the message located in the `x.cat` message source file that has the ID number of `1` and the set number of `2`:

```
dspmsg x.cat -s 2 1
```

You can use the **dspmsg** command in shell scripts when a message must be obtained from a message catalog.

Displaying Messages with an Application Program

When programming with the Message Facility, you must include the following items in your application program:

- The *CatalogFile*_**msg.h** definition file created by the **mkcatdefs** or **runcat** command if you used symbolic identifiers in the message source file, or the **limits.h** and **nl_types.h** files if you did not use symbolic identifiers.
- A call to initialize the locale environment.
- A call to open a catalog.
- A call to read a message.
- A call to display a message.
- A call to close the catalog.

The following subroutines provide the services necessary for displaying program messages with the message facility:

setlocale	Sets the locale. Specify the **LC_ALL** or **LC_MESSAGES** environment variable in the call to the **setlocale** subroutine for the preferred message catalog language.
catopen	Opens a specified message catalog and returns a catalog descriptor, which you use to retrieve messages from the catalog.
catgets	Retrieves a message from a catalog after a successful call to the **catopen** subroutine.
printf	Converts, formats, and writes to the stdout (standard output) stream.
catclose	Closes a specified message catalog.

The following C program, `hello`, illustrates opening the `hello.cat` catalog with the **catopen** subroutine, retrieving messages from the catalog with the **catgets** subroutine, displaying the messages with the **printf** subroutine, and closing the catalog with the **catclose** subroutine.

```
/* program: hello */
#include <nl_types.h>
#include <locale.h>
nl_catd catd;
main()
{
/*  initialize the locale  */
setlocale (LC_ALL, "");
/* open the catalog */
catd=catopen("hello.cat",NL_CAT_LOCALE);
printf(catgets(catd,1,1,"Hello World!"));
catclose(catd);                    /* close the catalog */
exit(0);
}
```

In the previous example, the **catopen** subroutine refers to the `hello.cat` message catalog only by file name. Therefore, you must make sure that the **NLSPATH** environment variable is set correctly. If the message catalog is successfully opened by the **catopen** subroutine, the **catgets** subroutine returns a pointer to the specified message in the `hello.cat` catalog. If the message catalog is not found or the message does not exist in the catalog, the **catgets** subroutine returns the `Hello World!` default string.

Understanding the NLSPATH Environment Variable

The **NLSPATH** environment variable specifies the directories to search for message catalogs. The **catopen** subroutine searches these directories in the order specified when called to locate and open a message catalog. If the message catalog is not found, the message-retrieving routine returns the program-supplied default message. See the **/etc/environment** file for the **NLSPATH** default path.

Retrieving Program-Supplied Default Messages

All message-retrieving routines return the program-supplied default message text if the desired message cannot be retrieved for any reason. Program-supplied default messages are generally brief one-line messages that contain no message numbers in the text. Users who prefer these default messages can set the **LC_MESSAGES** category to the C locale or unset the **NLSPATH** environment variable. When none of the **LC_ALL**, **LC_MESSAGES**, or **LANG** environment variables are set, the **LC_MESSAGES** category defaults to the C locale.

Setting the Language Hierarchy

Multilingual users may specify a language hierarchy for message text. To set the language hierarchy for the system default or for an individual user, see the **chlang** command, ″Changing the Language Environment″ in *AIX 5L Version 5.1 System Management Guide: Operating System and Devices* or use the System Management Interface Tool (SMIT). To use SMIT to set the language hierarchy, enter the SMIT fast path

```
smit mlang
```

at the command line.

Select **Change / Show Language Hierarchy**

OR

At the command line, enter:

```
smit
```

Select **System Environments**

Select **Manage Language Environment**

Select **Change / Show Language Hierarchy**

Example of Retrieving a Message from a Catalog

This example has three parts: the message source file, the command used to generate the message catalog file, and an example program using the message catalog.

1. The following example shows the `example.msg` message source file:

   ```
   $quote "
   $ every message catalog should have a beginning set number.
   $set MS_SET1
   MSG1 "Hello world\n"
   MSG2 "Good Morning\n"
   ERRMSG1 "example: 1000.220 Read permission is denied for the file
   %s.\n"
   $set MS_SET2
   MSG3 "Howdy\n"
   ```

2. The following command uses the `example.msg` message source file to generate the `example.h` header file and the `example.cat` catalog file in the current directory:

   ```
   runcat example example.msg
   ```

3. The following example program uses the `example.h` header file and accesses the `example.cat` catalog file:

   ```
   #include <locale.h>
   #include <nl_types.h>
   #include "example_msg.h" /*contains definitions for symbolic
                             identifiers*/
   main()
   ```

```
{
        nl_catd catd;
        int error;

        (void)setlocale(LC_ALL, "");

        catd = catopen(MF_EXAMPLE, NL_CAT_LOCALE);
        /*
        ** Get the message number 1 from the first set.
        */
        printf( catgets(catd,MS_SET1,MSG1,"Hello world\n") );

        /*
        ** Get the message number 1 from the second set.
        */
        printf( catgets(catd, MS_SET2, MSG3,"Howdy\n") );
        /*
        ** Display an error message.
        */
        printf( catgets(catd, MS_SET1, ERRMSG1,"example: 100.220
                Permission is denied to read the file %s.\n") ,
                filename);
        catclose(catd);
}
```

Culture-Specific Data Processing

Culture-specific data handling may be part of a program, and such a program may supply different data for different locales. In addition, a program may use different algorithms to process character data based on the language and culture. For example, recognition of the start and end of a word and the method of hyphenation of a word across two lines varies depending on the locale. Programs that deal with such functionality need access to these tables or algorithms based on the current locale setting at runtime. You can handle such programs in the following ways:

- Compile all the algorithms and tables, and load them with the program.

  This makes it difficult to add or modify the algorithms and tables. Whenever a new algorithm or table is added, the entire program must be relinked.
- Keep the locale-specific algorithms and tables in a file, and load them at run time depending on the current locale setting.

  This makes it easier to modify and add algorithms and tables. However, there is no standard defined way to load algorithms. In AIX, you can achieve this using the **load** system call, but programs that use the **load** system call may not be portable to other systems.

Culture-Specific Tables

If the culture-specific data can be processed by accessing tables based on the current locale setting, then this can be accomplished by using the standard file I/O subroutines (**fopen**, **fread**, **open**, **read**, and so on). Such tables must be provided in the directory defined in **/usr/lpp/***Name* where *Name* is the name of the particular application under the appropriate locale name.

Standard path prefix:	**/usr/lpp/***Name* (AIX-specific pathname)
Culture-specific directory:	Obtain the current locale for the appropriate category that describes the tables. Concatenate it to the above prefix.
Access:	Use standard file access subroutines (**fopen**, **fread**, and so on) as appropriate.

Culture-Specific Algorithms

The culture-specific algorithms reside in the **/usr/lpp**/*Name*/%**L** directory. Here %**L** represents the current locale setting for the appropriate category.

Use the **load** system call to access program-specific algorithms from an object module.

Standard path prefix:	**/usr/lpp/***Name*
Culture-specific directory:	Obtain the current locale for the appropriate category. Concatenate it to the above prefix.
Method:	Concatenate the method name to it.

Example: Load a Culture-Specific Module for Arabic Text for an Application

Header File

The `methods.h` include file has one structure as follows:

```
struct Methods {
        int     version;
        char    *(*hyphen)();
        char    *(*wordbegin)();
        char    *(*wordend)();
} ;
```

The Main Program

Let the program name be `textpr`.

The main program determines the module to load and invokes it. Note that the `textpr.h` include file is used to specify the path name of the load object. This way, the path name, which is system-specific, can be changed easily.

```
#include    <stdio.h>
#include    <errno.h>
#include    "methods.h"
#include    "textpr.h"    /* contains the pathname where
                             the load object can be found */

extern     int   errno;

main()
{
     char libpath[PATH_MAX];   /* stores the full pathname of the
                                  load object */
      char *prefix_path=PREFIX_PATH;        /* from textpr.h */
      char *method=METHOD;                  /* from textpr.h */
      int (*func)();
      char *path;
      /* Methods */
      int   ver;
      char  *p;
      struct Methods *md;


      setlocale(LC_ALL, "");
      path = setlocale(LC_CTYPE, 0);    /* obtain the locale
                                           for LC_CTYPE category */
      /* Construct the full pathname for the */
      /* object to be loaded                 */
      strcpy(libpath, prefix_path);
      strcat(libpath, path);
      strcat(libpath, "/");
      strcat(libpath, method);

      func = load(conv, 1, libpath);        /* load the object */
      if(func==NULL){
            strerror(errno);
            exit(1);
      }
      /* invoke the loaded module ");
```

```
      md =(struct Methods *) func();       /* Obtain the methods
                                              structure */
      ver = md->version;
      /* Invoke the methods as needed */
      p = (md->hyphen)();
      p = (md->wordbegin)();
      p = (md->wordend)();
}
```

Methods

This module contains culture-specific algorithms. In this example, it provides the Arabic method. The `method.c` program follows:

```
#include "methods.h"

char *Arabic_hyphen(char *);
char *Arabic_wordbegin(char *);
char *Arabic_wordend(char *);

static struct Methods ArabicMethods= {
        1,
        Arabic_hyphen,
        Arabic_wordbegin,
        Arabic_wordend
} ;

struct Methods *start_methods()
{
        /* startup methods */
        return ( &ArabicMethods);
}

char *Arabic_hyphen(char *string)
{
        /* Arabic hyphen */
        return( string );
}
char *Arabic_wordbegin(char *string)
{
        /*Arabic word begin */);
        return( string );
}
char *Arabic_wordend(char *string)
{
        /* Arabic word end */;
        return( string);
}
```

Include File: textpr

The include file contains the path name of the module to be loaded.

```
#define PREFIX_PATH "/usr/lpp/textpr"
                    /* This is an AIX-specific pathname */
```

NLS Sample Program

This sample program fragment, `foo.c`, illustrates internationalization through code set independent programming.

Message Source File for foo

A sample message source file for the `foo` utility is given here. Note we defined only one set and three messages in this catalog for illustration purposes only. A typical catalog contains several such messages.

The following is the message source file for `foo`, `foo.msg`.

```
$quote "

$set MS_FOO
```

```
CANTOPEN        "foo: cannot open %s\n"
BYTECNT         "number of bytes: %d\n"
CHARCNT         "number of characters: %d
```

Creation of Message Header File for foo

To generate the run-time catalog, use the **runcat** command as follows:

```
runcat foo foo.msg
```

This generates the header file `foo_msg.h` as shown in the following section. Note that the set mnemonic is `MS_FOO` and the message mnemonics are `CANTOPEN`, `BYTECNT`, and `CHARCNT`. These mnemonics are used in the programs on the following pages.

```
/*
** The header file: foo_msg.h is as follows:
*/
#ifndef _H_FOO_MSG
#define _H_FOO_MSG
#include <limits.h>
#include <nl_types.h>
#define MF_FOO "foo.cat"

/* The following was generated from wc.msg. */


/* definitions for set MS_FOO */
#define MS_FOO 1

#define CANTOPEN 1
#define BYTECNT  2
#define CHARCNT  3

#endif
```

Single Path Code Set Independent Version

The term *single source single path* refers to one path in a single application to be used to process both single-byte and multibyte code sets. The single source single path method eliminates all **ifdef**s for internationalization. All characters are handled the same way whether they are members of single-byte or multibyte code sets.

Single source single path is desirable but may degrade performance. Thus, it is not recommended for all programs. There may be some programs that do not suffer any performance degradation when they are fully internationalized; in those cases, use the single source single path method.

The following fully internationalized version of the `foo` utility supports all code sets through single source single path, code-set independent programming:

```
/*
 * COMPONENT_NAME:
 *
 * FUNCTIONS: foo
 *
 * The following code shows how to count the number of bytes and
 * the number of characters in a text file.
 *
 * This example is for illustration purposes only. Performance
 * improvements may still be possible.
 *
 */
```

```
#include        <stdio.h>
#include        <ctype.h>
#include        <locale.h>
#include        <stdlib.h>
#include        "foo_msg.h"

#define MSGSTR(Num,Str) catgets(catd,MS_FOO,Num,Str)

/*
 * NAME: foo
 *
 * FUNCTION: Counts the number of characters in a file.
 *
 */

main(argc,argv)
int argc;
char **argv;
{
    int     bytesread,   /* number of bytes read */
        bytesprocessed;
    int     leftover;

    int     i;
    int     mbcnt;           /* number of bytes in a character */
    int     f;               /* File descriptor */
    int mb_cur_max;
    int    bytect;           /* name changed from charct... */
    int    charct;           /* for real character count */
    char   *curp, *cure;     /* current and end pointers into
                              ** buffer */
    char         buf[BUFSIZ+1];

    nl_catd      catd;

    wchar_t    wc;

    /* Obtain the current locale */
    (void) setlocale(LC_ALL,"");

    /* after setting the locale, open the message catalog */
    catd = catopen(MF_FOO,NL_CAT_LOCALE);

    /* Parse the arguments if any */

    /*
    ** Obtaint he maximum number of bytes in a character in the
    ** current locale.
    */
    mb_cur_max = MB_CUR_MAX;
    i = 1;

    /* Open the specified file and issue error messages if any */
    f = open(argv[i],0);
    if(f<0){
        fprintf(stderr,MSGSTR(CANTOPEN,              /*MSG*/
            "foo: cannot open %s\n"), argv[i]);      /*MSG*/
            exit(2);
    }

    /* Initialize the variables for the count */
    bytect = 0;
    charct = 0;

    /* Start count of bytes and characters  */

    leftover = 0;

    for(;;) {
        bytesread = read(f,buf+leftover, BUFSIZ-leftover);
        /* issue any error messages here, if needed */
        if(bytesread <= 0)
             break;
```

```
        buf[leftover+bytesread] = '\0';
                /* Protect partial reads */
        bytect += bytesread;
        curp=buf;
        cure = buf + bytesread+leftover;
        leftover=0;      /* No more leftover */
        for(; curp<cure ;){
            /* Convert to wide character */
            mbcnt= mbtowc(&wc, curp, mb_cur_max);
            if(mbcnt <= 0){
                mbcnt = 1;
            }else if (cure - curp >=mb_cur_max){
                wc = *curp;
                mbcnt =1;
            }else{
                /* Needs more data */
                leftover= cure - curp;
                strcpy(buf, curp, leftover);
                break;
            }
            curp +=mbcnt;
            charct++;
        }
    }
        /* print number of chars and bytes */
    fprintf(stderr,MSGSTR(BYTECNT, "number of bytes:%d\n"),
            bytect);
    fprintf(stderr,MSGSTR(CHARCNT, "number of characters:%d\n"),
            charct);
    close(f);
    exit(0);
```

Dual-Path Version Optimized for Single-Byte Code Sets

The term *single source dual path* refers to two paths in a single application where one of the paths is chosen at run time depending on the current locale setting, which indicates whether the code set in use is single-byte or multibyte.

If a program can retain its performance and not increase its executable file size too much, the single source dual path method is the preferred choice. You should evaluate the increase in the executable file size on a per command or utility basis.

In the single byte dual path method, the **MB_CUR_MAX** macro specifies the maximum number of bytes in a multibyte character in the current locale. This should be used to determine at run time whether the processing path to be chosen is the single-byte or the multibyte path. Use a boolean flag to indicate the path to be chosen, for example:

```
int mbcodeset ;
/* After setlocale(LC_ALL,"") is done, determine the path to
** be chosen.
*/
if(MB_CUR_MAX == 1)
        mbcodeset = 0;
else    mbcodeset = 1;
```

This way, the current code set is checked to see if it is a multibyte code set and if so, the flag `mbcodeset` is set appropriately. Testing this flag has less performance impact than testing the **MB_CUR_MAX** macro several times.

```
if(mbcodeset){
        /* Multibyte code sets (also supports single-byte
        ** code sets )
        */
```

```
        /* Use multibyte or wide character processing
        functions */
}else{
        /* single-byte code sets */
        /* Process accordingly */
}
```

This approach is appropriate if internationalization affects a small proportion of a module. Excessive tests for providing dual paths may degrade performance. Provide the test at a level that precludes frequent testing for this case.

This following version of the `foo` utility produces one object, yet at run time the appropriate path is chosen based on the code set to optimize performance for that code set. Note we distinguish between single and multibyte code sets only.

```
/*
 * COMPONENT_NAME:
 *
 * FUNCTIONS: foo
 *
 * The following code shows how to count the number of bytes and
 * the number of characters in a text file.
 *
 * This example is for illustration purposes only. Performance
 * improvements may still be possible.
 *
 */

#include        <stdio.h>
#include        <ctype.h>
#include        <locale.h>
#include        <stdlib.h>
#include        "foo_msg.h"

#define MSGSTR(Num,Str) catgets(catd,MS_FOO,Num,Str)

/*
 * NAME: foo
 *
 * FUNCTION: Counts the number of characters in a file.
 *
 */

main(argc,argv)
int argc;
char **argv;
{
    int bytesread,  /* number of bytes read */
        bytesprocessed;
    int   leftover;

    int   i;
    int   mbcnt;   /* number of bytes in a character */
    int   f;       /* File descriptor */
    int   mb_cur_max;
    int    bytect;             /* name changed from charct... */
    int    charct;             /* for real character count */
    char   *curp, *cure; /* current and end pointers into buffer                          */
    char        buf[BUFSIZ+1];

        nl_catd          catd;

        wchar_t    wc;

        /* flag to indicate if current code set is a
        ** multibyte code set
        */
        int     multibytecodeset;

        /* Obtain the current locale */
        (void) setlocale(LC_ALL,"");
```

```
/* after setting the locale, open the message catalog */
catd = catopen(MF_FOO,NL_CAT_LOCALE);

/* Parse the arguments if any */

/*
** Obtain the maximum number of bytes in a character in the
** current locale.
*/
mb_cur_max = MB_CUR_MAX;

if(mb_cur_max >1)
    multibytecodeset = 1;
else
    multibytecodeset = 0;

i = 1;

/* Open the specified file and issue error messages if any */
f = open(argv[i],0);
if(f<0){
    fprintf(stderr,MSGSTR(CANTOPEN,              /*MSG*/
        "foo: cannot open %s\n"), argv[i]);      /*MSG*/
        exit(2);
}

/* Initialize the variables for the count */
bytect = 0;
charct = 0;

/* Start count of bytes and characters  */

leftover = 0;

if(multibytecodeset){
    /* Full internationalzation */
    /* Handles supported multibyte code sets */
    for(;;) {
        bytesread = read(f,buf+leftover,
                BUFSIZ-leftover);
        /* issue any error messages here, if needed */
        if(bytesread <= 0)
            break;

        buf[leftover+bytesread] = '\0';
                /* Protect partial reads */
        bytect += bytesread;
        curp=buf;

        cure = buf + bytesread+leftover;
        leftover=0; /* No more leftover */

        for(; curp<cure ;){
            /* Convert to wide character */
            mbcnt= mbtowc(&wc, curp, mb_cur_max);
            if(mbcnt <= 0){
                mbcnt = 1;
            }else if (cure - curp >=mb_cur_max){
                wc = *curp;
                mbcnt =1;

            }else{
                /* Needs more data */
                leftover= cure - curp;
                strcpy(buf, curp, leftover);
                break;
            }
            curp +=mbcnt;
            charct++;
        }
    }
}else {
```

```
    /* Code specific to single-byte code sets that
    ** avoids conversion to widechars and thus optimizes
    ** performance for single-byte code sets.
    */
    for(;;) {
        bytesread = read(f,buf, BUFSIZ);
        /* issue any error messages here, if needed */
        if(bytesread <= 0)
                break;
            bytect += bytesread;
            charct += bytesread;

    }
}
    /* print number of chars and bytes */
fprintf(stderr,MSGSTR(BYTECNT, "number of bytes:%d\n"),
        bytect);
fprintf(stderr,MSGSTR(CHARCNT, "number of characters:%d\n"),
        charct);
close(f);
exit(0);
```

National Language Support (NLS) Quick Reference

The NLS Quick Reference provides a place to get started internationalizing programs. The following sections offer advice and a practical guide through the NLS documentation:

- "National Language Support Do's and Don'ts" lists NLS guiding principles.
- "National Language Support Checklist" on page 498 provides a way to analyze a program for NLS dependencies.
- "Message Suggestions" on page 499 lists some guidelines for creating clearer and maintainable messages.

National Language Support Do's and Don'ts

The following list presents a set of NLS guiding principles and advice. The intention is to prevent the occurrence of common errors when internationalizing programs. See "Chapter 16. National Language Support" on page 329 for more information about NLS.

- DO externalize any user and error messages. We recommend the use of message catalogs. X applications may use resource files to externalize messages for each locale. See the "Message Facility Overview for Programming" on page 480 for more information.
- DO use standard X/Open, ISO/ANSI C, and POSIX functions to maximize portability. See ″NLS Subroutines Overview″ ("National Language Support Subroutines Overview" on page 339) for more information.
- DO use the font set specification in order to be code-set independent in X applications.
- DO use Xm (Motif) library widgets for building bidirectional and character shaping applicaitons. See ″Layout (Bidirectional) Support in Xm (Motif) Library″ in *AIX 5L Version 5.1 AIXwindows Programming Guide* for general information. Refer to the **XmText** or **XmTextField** widgets for support of input and output of bidirectional and shaping characteristics.
- DON'T assume the size of all characters to be 8 bits, or 1 byte. Characters may be 1, 2, 3, 4 or more bytes. See "Multibyte Code and Wide Character Code Conversion Subroutines" on page 348 and the "Code Set Overview" on page 379 for more information.
- DON'T assume the encoding of any code set. See the "Code Set Overview" on page 379 for more information.
- DON'T hard code names of code sets, locales, or fonts because it may impact portability. See "Chapter 16. National Language Support" on page 329 for more information.

- DON'T use p++ to increment a pointer in a multibyte string. Use the **mblen** subroutine to determine the number of bytes that compose a character.
- DON'T assume any particular physical keyboard is in use. Use an input method based on the locale setting to handle keyboard input. See the "Input Method Overview" on page 452 for more information.
- DON'T define your own converter unless absolutely necessary. See the "Converters Overview for Programming" on page 410 for more information.
- DON'T assume that the **char** data type is either signed or unsigned. This is platform-specific. If the particular system that is used defines **char** to be **signed**, comparisons with full 8-bit quantity will yield incorrect results. As all the 8-bits are used in encoding a character, be sure to declare **char** as **unsigned char** wherever necessary. Also, note that if a **signed char** value is used to index an array, it may yield incorrect results. To make programs portable, define 8-bit characters as **unsigned char**.
- DON'T use the layout subroutines in the **libi18n.a** library unless the application is doing presentation types of services. Most applications just deal with logically ordered text. See "Introducing Layout Library Subroutines" on page 377 for more information.

National Language Support Checklist

The National Language Support (NLS) Checklist provides a way to analyze a program for NLS dependencies. By going through this list, one can determine what, if any, NLS functions must be considered. This is useful for both programming and testing. If you identify a set of NLS items that a program depends on, a test strategy can be developed. This facilitates a common approach to testing all programs.

All major NLS considerations have been identified. However, this list is not all-encompassing. There may be other NLS questions that are not listed. See "Chapter 16. National Language Support" on page 329 for more information about NLS. See "National Language Support Do's and Don'ts" on page 497 for a brief list of NLS advice.

AIXwindows CheckList

The remaining checklist items are specific to the AIXwindows systems.

1. Does your client use labels, buttons, or other output-only widgets to display translatable messages?

   If yes:

   - Invoke the ***XtSetLanguageProc** subroutine in the following manner:

     ```
     XtSetLanguageProc(NULL, NULL, NULL);
     ```

   - Messages can be placed in either message catalogs or localized resource files. See checklist items 1 or 20, respectively.
   - To make the widgets code set-independent, specify fonts that use font sets.
2. Does your client use X resource files to define the text of labels, buttons, or text widgets?

   If yes:

   - Put all resources that need translation in one place.
   - Consider using message catalogs for the text strings. See the "Message Facility Overview for Programming" on page 480 for more information.
   - Do not use translated color names, since color names are restricted to one encoding. The only portable names are encoded in the portable character set.
   - Put language-specific resource files in **/usr/lib/X11/%L/app-defaults/%N**, where **%L** is the name of the locale, such as fr_FR, and **%N** is the name of the client.
3. Is keyboard input localized by language?

   If yes:

   - Invoke the ***XtSetLanguageProc** subroutine in the following manner:

     ```
     XtSetLanguageProc(NULL, NULL, NULL);
     ```

   - Use the **XmText** or **XmTextField** widgets for all text input.

Some of the **XmText** widgets' arguments are defined in terms of character length instead of byte length. The cursor position is maintained in character position, not byte position.

- Are you using the **XmDrawingArea** widget to do localized input?
  - Use the input method subroutines to do input processing in different languages. See the "Input Method Overview" on page 452 and the **IMAuxDraw** Callback subroutine for more information.

4. Does your client present lists or labels consisting of localized text from user files rather than from X resource files?

   If yes:

   - Invoke the ***XtSetLanguageProc** subroutine in the following manner:

     ```
     XtSetLanguageProc(NULL, NULL, NULL);
     ```

   - Use the **XmStringCreateSimple** subroutine to create the **XmString** data type for localized text. The **XmStringCreate** subroutine can be used, but **XmSTRING_DEFAULT_CHARSET** is preferable.
   - To make the widgets code-set independent, specify fonts by using font sets. Font resources (for example, ***fontList:** instead) in the app-defaults files should use the upper case and class form rather than the lower case form (for example, ***FontList:** instead). This allow the desktop style manager to affect the application font selection.

5. Does your program do any presentation operations (Xlib drawing, printing, formatting, or editing) on bidirectional text?

   If yes:

   - Use the **XmText** or **XmTextField** in the Xm (Motif) library. These widgets are enabled for bidirectional text. See "Layout (Bidirectional) Support in Xm (Motif) Library" in *AIX 5L Version 5.1 AIXwindows Programming Guide* for more information.
   - If the Xm library can not be used, use the layout subroutines to perform any re-ordering and shaping on the text. See "Introducing Layout Library Subroutines" on page 377 for more information.
   - Store and communicate the text in the implicit (logical) form. Some utilities (for example, **aixterm**) support the visual form of bidirectional text, but most NLS subroutines can not process the visual form of bidirectional text.

If the response to all the above items is no, then the program probably has no NLS dependencies. In this case, you may not need the locale-setting subroutine **setlocale** and the catalog facility subroutines **catopen** and **catgets**.

Message Suggestions

The following are suggestions on how to make messages meaningful and concise:

- Plan for the translation of all messages, including messages that are displayed on panels.
- Externalize messages.
- Provide default messages.
- Make each message in a message source file be a complete entity. Building a message by concatenating parts together makes translation difficult.
- Use the **$len** directive in the message source file to control the maximum display length of the message text. (The **$len** directive is specific to the Message Facility.)
- Allow sufficient space for translated messages to be displayed. Translated messages often occupy more display columns than the original message text. In general, allow about 20% to 30% more space for translated messages, but in some cases you may need to allow 100% more space for translated messages.
- Use symbolic identifiers to specify the set number and message number. Programs should refer to set numbers and message numbers by their symbolic identifiers, not by their actual numbers. (The use of symbolic identifiers is specific to the Message Facility.)

- Facilitate the reordering of sentence clauses by numbering the %**s** variables. This allows the translator to reorder the clauses if needed. For example, if a program needs to display the English message: `The file %s is referenced in %s`, a program may supply the two strings as follows:

  ```
  printf(message_pointer, name1, name2)
  ```

  The English message numbers the %**s** variables as follows:

  ```
  The file %1$s is referenced in %2$s\n
  ```

  The translated equivalent of this message may be:

  ```
  %2$s contains a reference to file %1$s\n
  ```

- Do not use **sys_errlist[errno]** to obtain an error message. This defeats the purpose of externalizing messages. The **sys_errlist[]** is an array of error messages provided only in the English language. Use **strerror**(*errno*) , as it obtains messages from catalogs.
- Do not use **sys_siglist[signo]** to obtain an error message. This defeats the purpose of externalizing messages. The **sys_siglist[]** is an array of error messages provided only in the English language. Use **psignal() ,** as it obtains messages from catalogs.
- Use the message comments facility to aid in the maintenance and translation of messages.
- In general, create separate message source files and catalogs for messages that apply to each command or utility.

Describing Command Syntax in Messages

- Show the command syntax in the usage statement. For example, a possible usage statement for the **rm** command is:

  ```
  Usage: rm [-firRe] [--] File ...
  ```

- Capitalize the first letter of such words as File, Directory, String, and Number in usage statement messages.
- Do not abbreviate parameters on the command line. For example, `Num` spelled out `as Number` can be more easily translated.
- Use only the following delimiters in usage statement messages:

[]	Encloses an optional parameter.
{}	Encloses multiple parameters, one of which is required.
|	Seperates parameters that cannot both be chosen. For example, `[a|b]` indicates that you can choose `a`, `b` , or neither `a` nor `b` ; and `{a|b}` indicates that you must choose `a` or `b` .
...	Follows a parameter that can be repeated on the command line. Note that there is a space before the ellipsis.
-	Indicates standard input.

- Do not use any delimiters for a required parameter that is the only choice. For example:

  ```
  banner String
  ```

- Put a space character between flags that must be separated on the command line. For example:

  ```
  unget [-n] [-rSID] [-s] {File|-}
  ```

- Do not separate flags that can be used together on the command line. For example:

  ```
  wc [-cwl] {File ...|-}
  ```

- Put flags in alphabetical order when the order of the flags on the command line does not make a difference. Put lowercase flags before uppercase flags. For example:

  ```
  get -aAijlmM
  ```

- Use your best judgment to determine where you should end lines in the usage statement message. The following example shows a lengthy usage statement message:

  ```
  Usage: get [-e|-k] [-cCutoff] [-iList] [-rSID] [-wString] [-xList] [-b] [-gmnpst] [-l[p]] File ...
  ```

Writing Style of Messages

Clear writing aids in message translation. The following guidelines on the writing style of messages include terminology, punctuation, mood, voice, tense, capitalization, format, and other usage questions.

- Write concise messages. One-sentence messages are preferable.
- Use complete-sentence format.
- Add articles (a, an, the) when necessary to eliminate ambiguity.
- Capitalize the first word of the sentence, and use a period at the end of the sentence.
- Use the present tense. Do not use future tense in a message. For example, use the sentence:

  `The cal command displays a calendar.`

  Instead of:

  `The cal command will display a calendar.`
- Do not use the first person (I or we) in messages.
- Avoid using the second person (you) except in help and interactive text.
- Use active voice. The following example shows how a message written in passive voice can be turned into an active voice message.

  Passive: `Month and year must be entered as numbers.`
  Active: `Enter month and year as numbers.`
- Use the imperative mood (command phrase) and active verbs such as specify, use, check, choose, and wait.
- State messages in a positive tone. The following example shows a negative message made more positive.

  Negative: `Don't use the f option more than once.`
  Positive: `Use the -f flag only once.`
- Use words only in the grammatical categories shown in a dictionary. If a word is shown only as a noun, do not use it as a verb. For example, do not *solution* a problem or *architect* a system.
- Do not use prefixes or suffixes. Translators may not know what words beginning with re-, un-, in-, or non- mean, and the translations of messages that use prefixes or suffixes may not have the meaning you intended. Exceptions to this rule occur when the prefix is an integral part of a commonly used word. For example, the words `previous` and `premature` are acceptable; the word `nonexistent` is not acceptable.
- Do not use parentheses to show singular or plural, as in `error(s)`, which cannot be translated. If you must show singular and plural, write *error or errors*. You may also be able to revise the code so that different messages are issued depending on whether the singular or plural of a word is required.
- Do not use contractions.
- Do not use quotation marks, both single and double quotation marks. For example, do not use quotation marks around variables such as `%`**s**, `%`**c**, and `%`**d** or around commands. Users may interpret the quotation marks literally.
- Do not hyphenate words at ends of lines.
- Do not use the standard highlighting guidelines in messages, and do not substitute initial or all caps for other highlighting practices. (Standard highlighting includes such guidelines as bold for commands, subroutines, and files; italics for variables and parameters; typewriter or courier for examples and displayed text.)
- Do not use the and/or construction. This construction does not exist in other languages. Usually it is better to say `or` to indicate that it is not necessary to do both.
- Use the 24-hour clock. Do not use a.m. or p.m. to specify time. For example, write `1:00 p.m.` as `1300`.
- Avoid acronyms. Only use acronyms that are better known to your audience than their spelled-out version. To make a plural of an acronym, add a lowercase s without an apostrophe. Verify that the acronym is not a trademark before using it.

- Do not construct messages from clauses. Use flags or other means within the program to pass on information so that a complete message may be issued at the proper time.
- Do not use hard-coded text as a variable for a **%s** string in a message.
- End the last line of the message with **\n** (indicating a new line). This applies to one-line messages also.
- Begin the second and remaining lines of a message with **\t** (indicating a tab).
- End all other lines with **\n** (indicating a new line).
- Force a newline on word boundaries where needed so that acceptable message strings display. The **printf** subroutine, which often is used to display the message text, disregards word boundaries and wraps text whenever necessary, sometimes splitting a word in the middle.
- If, for some reason, the message should not end with a newline character, leave writers a comment to that effect.
- Precede each message with the name of the command that called the message, followed by a colon. The following example is a message containing a command name:

  ```
  OPIE "foo: Opening the file."
  ```
- Tell the user to `Press the —— key` to select a key on the keyboard, including the specific key to press. For example:

  ```
  Press the Ctrl-D key
  ```
- Do not tell the user to `Try again later`, unless the system is overloaded. The need to try again should be obvious from the message.
- Use the word ″parameter″ to describe text on the command line, the word ″value″ to indicate numeric data, and the words ″command string″ to describe the command with its parameters.
- Do not use commas to set off the one-thousandth place in values. For example, use `1000` instead of `1,000`.
- If a message must be set off with an `*` (asterisk), use two asterisks at the beginning of the message and two at asterisks at the end of the message. For example:

  ```
  ** Total **
  ```
- Use the words ″log in″ and ″log off″ as verbs. For example:

  ```
  Log in to the system; enter the data; then log off.
  ```
- Use the words ″user name,″ ″group name,″ and ″login″ as nouns. For example:

  ```
  The user is sam.
  The group name is staff.
  The login directory is /u/sam.
  ```
- Do not use the word ″superuser.″ Note that the root user may not have all privileges.
- Use the following frequently occurring standard messages where applicable:

Preferred Standard Message	Less Desirable Message
`Cannot find or open the file.`	Can’t open filename.
`Cannot find or access the file.`	Can’t access
`The syntax of a parameter is not valid.`	syntax error

List of National Language Support Subroutines

The National Language Support (NLS) subroutines are used for handling locale-specific information, manipulating wide characters and multibyte characters, and using regular expressions. The following functional lists of NLS subroutines are provided:

- “List of Locale Subroutines” on page 503
- “List of Multibyte Character Subroutines” on page 503
- “List of Wide Character Subroutines” on page 503
- “List of Layout Library Subroutines” on page 505

- “List of Message Facility Subroutines” on page 505
- “List of Converter Subroutines” on page 505
- “List of Input Method Subroutines” on page 506
- “List of Regular Expression Subroutines” on page 506

For more information about NLS subroutines see “National Language Support Subroutines Overview” on page 339.

List of Locale Subroutines

The following subroutines are provided to obtain and process locale-specific data:

localeconv	Retrieves locale-dependent conventions of a program locale.
nl_langinfo	Returns information on language or cultural area in a program locale.
rpmatch	Determines whether a response is affirmative or negative in the current locale.
setlocale	Changes or queries a program’s current locale.

For more information about locales and their databases see “Locale Overview for Programming” on page 330

For more NLS subroutines see “List of National Language Support Subroutines” on page 502.

List of Time and Monetary Formatting Subroutines

strfmon	Formats monetary strings according to the current locale.
strftime	Formats time and date according to the current locale.
strptime	Converts a character string to a time value according to the current locale.
wcsftime	Converts time and date into a wide character string according to the current locale.

For more information about NLS subroutines see “National Language Support Subroutines Overview” on page 339.

For more NLS subroutines see “List of National Language Support Subroutines” on page 502.

List of Multibyte Character Subroutines

mblen	Determines the length of a multibyte character.
mbstowcs	Converts a multibyte character string to a wide character string.
mbtowc	Converts a multibyte character to a wide character.

For more information about multibyte character subroutines see “National Language Support Subroutines Overview” on page 339.

For more NLS subroutines see “List of National Language Support Subroutines” on page 502.

List of Wide Character Subroutines

The following subroutines process characters in process-code form:

fgetwc	Gets a wide character or word from an input stream.
fgetws	Gets a wide character string from a stream.
fputwc	Writes a wide character or a word to a stream.

fputws	Writes a wide character string to a stream.
getwc	Gets a wide character or word from an input stream.
getwchar	Gets a wide character or word from an input stream.
getws	Gets a wide character string from a stream.
iswalnum	Determines if the wide character is alphanumeric.
iswalpha	Determines if the wide character is alphabetic.
iswcntrl	Determines if the wide character is a control character.
iswctype	Determines the property of a wide character.
iswdigit	Determines if the wide character is a digit.
iswgraph	Determines if the wide character (excluding ″space characters″) is a printing character.
iswlower	Determines if the wide character is lowercase.
iswprint	Determines if the wide character (including ″space characters″) is a printing character.
iswpunct	Determines if the wide character is a punctuation character.
iswspace	Determines if the wide character is a blank space.
iswupper	Determines if the wide character is uppercase.
iswxdigit	Determines if the wide character is a hexadecimal digit.
putwc	Writes a wide character or a word to a stream.
putwchar	Writes a wide character or a word to a stream.
putws	Writes a wide character string to a stream.
strcoll	Compares two strings based on their collation weights in the current locale.
strxfrm	Transforms a string into locale collation values.
towlower	Converts an uppercase wide character to a lowercase wide character.
towupper	Converts a lowercase wide character to an uppercase wide character.
ungetwc	Pushes a wide character onto a stream.
wcsid	Returns the charsetID of a wide character.
wcscat	Concatenates wide character strings.
wcschr	Searches for a wide character.
wcscmp	Compares wide character strings.
wcscoll	Compares the collation weights of wide character strings.
wcscpy	Copies a wide character string.
wcscspn	Searches for a wide character string.
wcslen	Determines the number of characters in a wide character string.
wcsncat	Concatenates a specified number of wide characters.
wcsncmp	Compares a specified number of wide characters.
wcsncpy	Copies a specified number of wide characters.
wcspbrk	Locates the first occurrence of wide characters in a wide character string.
wcsrchr	Locates the last occurrence of wide characters in a wide character string.
wcsspn	Returns the number of wide characters in the initial segment of a string.
wcstod	Converts a wide character string to a double-precision floating point value.
wcstok	Breaks a wide character string into a sequence of separate wide character strings.
wcstol	Converts a wide character string to a long integer value.
wcstombs	Converts a sequence of wide characters to a sequence of multibyte characters.
wcstoul	Converts a wide character string to an unsigned, long integer value.
wcswcs	Locates the first occurrence of a wide character sequence in a wide character string.
wcswidth	Determines the display width of a wide character string.
wcsxfrm	Converts a wide character string to values representing character collation weights.
wctomb	Converts a wide character to a multibyte character.
wctype	Gets a handle for valid property names as defined in the current locale.
wcwidth	Determines the display width of a wide character.

For more information about wide character subroutines see “National Language Support Subroutines Overview” on page 339.

For more NLS subroutines see “List of National Language Support Subroutines” on page 502.

List of Layout Library Subroutines

The following subroutines of the Layout library (**libi18n.a**) transform bidirectional and context-dependent text to different formats:

layout_object_create	Initializes a layout context.
layout_object_free	Frees a **LayoutObject** structure.
layout_object_editshape	Edits the shape of the context text.
layout_object_getvalue	Queries the current layout values of a **LayoutObject** structure.
layout_object_setvalue	Sets the layout values of a **LayoutObject** structure.
layout_object_shapeboxchars	Shapes box characters.
layout_object_transform	Transforms the text according to the current layout values of a **LayoutObject** structure.

For more information about Layout library subroutines see “National Language Support Subroutines Overview” on page 339.

For more NLS subroutines see “List of National Language Support Subroutines” on page 502.

List of Message Facility Subroutines

The Message Facility consists of standard defined subroutines and commands, and manufacturer value-added extensions to support externalized message catalogs. These catalogs are used by an application to retrieve and display messages, as needed. The following Message Facility subroutines get messages for an application:

catopen	Opens a catalog.
catgets	Gets a messages from a catalog.
catclose	Closes a catalog.
strerror	Maps an error number to an error-message string appropriate for the current locale.

For more information about multibyte character subroutines see “National Language Support Subroutines Overview” on page 339.

For more NLS subroutines see “List of National Language Support Subroutines” on page 502.

List of Converter Subroutines

In an internationalized environment, it is often necessary to convert data from one code set to another. The following converter subroutines are supported for this purpose:

iconv_open	Performs the initialization required to convert characters from the code set specified by the *FromCode* parameter to the code set specified by the *ToCode* parameter.
iconv	Invokes the converter function using the descriptor obtained from the **iconv_open** subroutine.
iconv_close	Closes the conversion descriptor specified by the *cd* variable and makes it usable again.
ccsidtocs	Returns the code-set name of the corresponding coded character set IDs (CCSID).
cstoccsid	Returns the CCSID of the corresponding code-set name.

For more information about multibyte character subroutines see “National Language Support Subroutines Overview” on page 339.

For more NLS subroutines see “List of National Language Support Subroutines” on page 502.

List of Input Method Subroutines

The Input Method is a set of subroutines that translate key strokes into character strings in the code set specified by a locale. The Input Method subroutines include logic for locale-specific input processing and keyboard controls (for example, Ctrl, Alt, Shift, Lock, and Alt-Graphic). The following subroutines support this Input Method:

IMAIXMapping	Translates a pair of *KeySymbol* and *State* parameters to a string and returns a pointer to that string.
IMAuxCreate	Tells the application program to create an auxiliary area.
IMAuxDestroy	Notifies the callback to destroy any knowledge of the auxiliary area.
IMAuxDraw	Tells the application program to draw the auxiliary area.
IMAuxHide	Tells the application program to hide the auxiliary area.
IMBeep	Tells the application program to emit a beep sound.
IMClose	Closes the input method.
IMCreate	Creates one instance of a particular input method.
IMDestroy	Destroys an input method instance.
IMFilter	Checks whether a keyboard event is used by the input method for its internal processing.
IMFreeKeymap	Frees resources allocated by the **IMInitialzieKeymap** subroutine.
IMIndicatorDraw	Tells the application program to draw the indicator.
IMIndicatorHide	Tells the application program to hide the indicator.
IMInitialize	Initializes the input method for a particular language.
IMInitializeKeymap	Initializes the input method for a particular language.
IMIoctl	Performs a variety of control or query operations on the input method.
IMLookupString	Maps a keyboard-symbol/state pair to a string defined by the user.
IMProcessAuxiliary	Notifies the input method of input for an auxiliary area.
IMQueryLanguage	Checks to see if the specified language is supported.
IMSimpleMapping	Translates a pair of *KeySymbol* and *State* parameters to a string a returns a pointer to that string.
IMTextCursor	Sets the new display cursor position.
IMTextDraw	Asks the application program to draw the next string.
IMTextHide	Tells the application program to hide the text area.
IMTextStart	Notifies the application program of the length of the pre-editing space.
IMTextStart	Notifies the application program of the length of the pre-editing space.

List of Regular Expression Subroutines

The following subroutines handle regular expressions:

regcomp	Compiles a regular expression for comparison by the **regexec** subroutine.

For more information about multibyte character subroutines see “National Language Support Subroutines Overview” on page 339.

For more NLS subroutines see “List of National Language Support Subroutines” on page 502.

Chapter 17. Object Data Manager (ODM)

Object Data Manager (ODM) is a data manager intended for storing system information. Information is stored and maintained as objects with associated characteristics. You can also use ODM to manage data for application programs.

System data managed by ODM includes:

- Device configuration information
- Display information for SMIT (menus, selectors, and dialogs)
- Vital product data for installation and update procedures
- Communications configuration information
- System resource information.

You can create, add, lock, store, change, get, show, delete, and drop objects and object classes with ODM. ODM commands provide a command line interface to these functions. ODM subroutines access these functions from within an application program.

Some object classes come with the system. These object classes are discussed in the documentation for the specific system products that provide them.

This chapter discusses:

ODM Object Classes and Objects

The basic components of ODM are object classes and objects. To manage object classes and objects, you use the ODM commands and subroutines ("List of ODM Commands and Subroutines" on page 517). Specifically, you use the create and add features of these interfaces to build object classes and objects for storage and management of your own data.

object class — A group of objects with the same definition. An object class comprises one or more descriptors ("ODM Descriptors" on page 510).

object — A member of a defined object class, is an entity that requires storage and management of data

An object class is conceptually similar to an array of structures, with each object being a structure that is an element of the array. Values are associated with the descriptors of an object when the object is added to an object class. The descriptors of an object and their associated values can be located and changed with ODM facilities.

The following example provides an overview of manipulating object classes and objects.

Example:

1. To create an object class called `Fictional_Characters`, enter:

```
class Fictional_Characters {
        char    Story_Star[20];
        char    Birthday[20];
        short   Age;
        char    Friend[20];
};
```

   In this example, the `Fictional_Characters` object class contains four descriptors: `Story_Star`, `Birthday`, and `Friend`, which have a descriptor type of character and a 20-character maximum length;

and Age, with a descriptor type of short. To create the object class files required by ODM, you process this file with the **odmcreate** command or the **odm_create_class** subroutine.

2. Once you create an object class, you can add objects to the class using the **odmadd** command or the **odm_add_obj** subroutine. For example, enter the following code with the **odmadd** command to add the objects Cinderella and Snow White to the Fictional_Characters object class, along with values for the descriptors they inherit:

```
Fictional_Characters:
        Story_Star      = "Cinderella"
        Birthday        = "Once upon a time"
        Age             = 19
        Friend          = "mice"

Fictional_Characters:
        Story_Star = "Snow White"
        Birthday = "Once upon a time"
        Age = 18
        Friend = "Fairy Godmother"
```

The Fictional_Characters table shows a conceptual picture of the Fictional_Characters object class with the two added objects Cinderella and Snow White.

Table 9. Conceptual Picture of Fictional_Characters Object Class with Two Objects, Cinderella and Snow White

Fictional Characters			
Story Star (char)	Birthday (char)	Age (short)	Friend (char)
Cinderella	Once upon a time	19	Mice
Snow White	Once upon a time	18	Fairy Godmother

```
Retrieved data for 'Story_Star = "Cinderella"'
     Cinderella:
          Birthday = Once upon a time
          Age = 19
          Friend = Mice
```

3. After the Fictional_Characters object class is created and the objects Cinderella and Snow White are added, the retrieved data for 'Story_Star = "Cinderella"' is:

```
Cinderella:
        Birthday        = Once upon a time
        Age             = 19
        Friend          = mice
```

Creating an Object Class

Prerequisite Tasks or Conditions

Attention: Making changes to files that define system object classes and objects can result in system problems. Consult your system administrator before using the **/usr/lib/objrepos** directory as a storage directory for object classes and objects.

1. Create the definition for one or more object classes in an ASCII file. “ODM Example Code and Output” on page 518 shows an ASCII file containing several object class definitions.
2. Specify the directory in which the generated object must be stored.

"ODM Object Class and Object Storage" discusses the criteria used at object-class creation time for determining the directory in which to store generated object classes and objects. Most system object classes and objects are stored in the **/usr/lib/objrepos** directory.

Procedure

Generate an empty object class by running the **odmcreate** command with the ASCII file of object class definitions specified as the *ClassDescriptionFile* input file.

Adding Objects to an Object Class

Prerequisite Tasks or Conditions

Attention: Making changes to files that define system object classes and objects can result in system problems. Consult your system administrator before using the **/usr/lib/objrepos** directory as a storage directory for object classes and objects.

1. Create the object class to which the objects will be added. See “Creating an Object Class” on page 508 for instructions on creating an object class.
2. Create the definitions for one or more objects. “ODM Example Code and Output” on page 518 shows an ASCII file containing several object definitions.
3. Specify the directory in which the generated objects will be stored.

″ODM Object Class and Object Storage″ discusses the criteria used at object class creation time for determining the directory in which to store generated object classes and objects. Most system object classes and objects are stored in the **/usr/lib/objrepos** directory.

Procedure

Add objects to an empty object class by running the **odmadd** command with the ASCII file of object definitions specified as the *InputFile* input file.

Locking Object Classes

ODM does not implicitly lock object classes or objects. The coordination of locking and unlocking is the responsibility of the applications accessing the object classes. However, ODM provides the **odm_lock** and **odm_unlock** subroutines to control locking and unlocking object classes by application programs.

odm_lock Processes a string that is a path name and can resolve in an object class file or a directory of object classes. It returns a lock identifier and sets a flag to indicate that the specified object class or classes defined by the path name are in use.

When the **odm_lock** subroutine sets the lock flag, it does not disable use of the object class by other processes. If usage collision is a potential problem, an application program should explicitly wait until it is granted a lock on a class before using the class.

Another application cannot acquire a lock on the same path name while a lock is in effect. However, a lock on a directory name does not prevent another application from acquiring a lock on a subdirectory or the files within that directory.

To unlock a locked object class, use an **odm_unlock** subroutine called with the lock identifier returned by the **odm_lock** subroutine.

Storing Object Classes and Objects

Each object class you create with an **odmcreate** command or **odm_create_class** subroutine is stored in a file as a C language definition of an array of structures. Each object you add to the object class with an **odmadd** command or an **odm_add_obj** subroutine is stored as a C language structure in the same file.

You determine the directory in which to store this file when you create the object class.

Prerequisite Tasks or Condition

Create an object or object class.

Procedure

Storage methods vary according to whether commands or subroutines are used to create object classes and objects.

> **Attention:** Making changes to files that define system object classes and objects can result in system problems. Consult your system administrator before using the **/usr/lib/objrepos** directory as a storage directory for object classes and objects.

Using ODM Commands

When using the **odmcreate** or **odmdrop** command to create or drop an object class, specify the directory from which the class definition file will be accessed as follows:

1. Store the file in the default directory indicated by **$ODMDIR**, which is the **/usr/lib/objrepos** directory.
2. Use the **set** command to set the **ODMDIR** environment variable to specify a directory for storage.
3. Use the **unset** command to unset the **ODMDIR** environment variable and the **cd** command to change the current directory to the one in which you want the object classes or objects stored. Then, run the ODM commands in that directory. The file defining the object classes and objects will be stored in the current directory.

When using the **odmdelete**, **odmadd**, **odmchange**, **odmshow**, or **odmget** command to work with classes and objects, specify the directory from which the class definition file will be accessed as follows:

1. Store the file in the default directory indicated by **$ODMDIR**, which is the **/usr/lib/objrepos** directory.
2. Use the **set** command to set the **ODMDIR** environment variable to specify a directory for storage.
3. Use the **unset** command to unset the **ODMDIR** environment variable and the **cd** command to change the current directory to the one in which you want the object classes or objects stored. Then, run the ODM commands in that directory. The file defining the object classes and objects will be stored in the current directory.
4. From the command line, use the **set** command to set the **ODMPATH** environment variable to a string containing a colon-separated list of directories to be searched for classes and objects. For example:

   ```
   $ export ODMPATH = /usr/lib/objrepos:/tmp/myrepos
   ```

   The directories in the **$ODMPATH** are searched only if the directory **$ODMDIR** does not have the class definition file.

Using the odm_create_class or odm_add_obj Subroutines

The **odm_create_class** or **odm_add_obj** subroutine is used to create object classes and objects:

- If there is a specific requirement for your application to store object classes other than specified by the **ODMDIR** environment variable, use the **odm_set_path** subroutine to reset the path. It is strongly recommended that you use this subroutine to set explicitly the storage path whenever creating object classes or objects from an application.

  OR
- Before running your application, use the **set** command from the command line to set the **ODMDIR** environment variable to specify a directory for storage.

  OR
- Store the file in the default object repository used to store most of the system object classes, the **/usr/lib/objrepos** directory.

ODM Descriptors

An Object Data Manager (ODM) descriptor is conceptually similar to a variable with a name and type. When an object class is created, its descriptors are defined like variable names with associated ODM descriptor types. When an object is added to an object class, it gets a copy of all of the descriptors of the object class. Values are also associated with object descriptors already stated.

ODM supports several descriptor types:

terminal descriptor ("ODM Terminal Descriptors")	Defines a character or numerical data type
link descriptor ("ODM Link Descriptor")	Defines a relationship between object classes
method descriptor ("ODM Method Descriptor" on page 513)	Defines an operation or method for an object

Use the descriptors of an object and their associated values to define criteria for retrieving individual objects from an object class. Format the selection criteria you pass to ODM as defined in "ODM Object Searches" on page 514. Do not use the **binary** terminal descriptor in search criteria because of its arbitrary length.

ODM Terminal Descriptors

Terminal descriptors define the most primitive data types used by ODM. A terminal descriptor is basically a variable defined with an ODM terminal descriptor type. The terminal descriptor types provided by ODM are:

short	Specifies a signed 2-byte number.
long	Specifies a signed 4-byte number.
ulong	Specifies an unsigned 4-byte number.
binary	Specifies a fixed-length bit string. The binary terminal descriptor type is defined by the user at ODM creation time. The binary terminal descriptor type cannot be used in selection criteria.
char	Specifies a fixed-length, null-terminated string.
vchar	Specifies variable-length, null-terminated string. The **vchar** terminal descriptor type can be used in selection criteria.
long64/ODM_LONG_LONG/int64	Specifies a signed 8-byte number.
ulong64/ODM_ULONG_LONG/uint64	Specifies an unsigned 8-byte number.

ODM Link Descriptor

The ODM *link descriptor* establishes a relationship between an object in an object class and an object in another object class. A link descriptor is a variable defined with the ODM link descriptor type.

For example, the following code can be processed by the ODM create facilities to generate the **Friend_Table** and **Fictional_Characters** object classes:

```
class Friend_Table {
        char    Friend_of[20];
        char    Friend[20];
};

  class Fictional_Characters {
        char    Story_Star[20];
        char    Birthday[20];
        short   Age;
        link    Friend_Table Friend_Table Friend_of Friends_of;
};
```

The `Fictional_Characters` object class uses a link descriptor to make the `Friends_of` descriptors link to the `Friend_Table` object class. To resolve the link, the `Friends_of` descriptor retrieves objects in the `Friend_Table` object class with matching data in its `Friend_of` descriptors. The link descriptor in the

Fictional_Characters object class defines the class being linked to (Friend_Table), the descriptor being linked to (Friend_of), and the name of the link descriptor (Friends_of) in the Fictional_Characters object class.

The following code could be used to add objects to the **Fictional_Characters** and **Friend_Table** object classes:

```
Fictional_Characters:
        Story_Star      = "Cinderella"
        Birthday        = "Once upon a time"
        Age             = 19
        Friends_of      = "Cinderella"
Fictional_Characters:
        Story_Star      = "Snow White"
        Birthday        = "Once upon a time"
        Age             = 18
        Friends_of      = "Snow White"
Friend_Table:
        Friend_of       = "Cinderella"
        Friend          = "Fairy Godmother"
Friend_Table:
        Friend_of       = "Cinderella"
        Friend          = "mice"
Friend_Table:
        Friend_of       = "Snow White"
        Friend          = "Sneezy"
Friend_Table:
        Friend_of       = "Snow White"
        Friend          = "Sleepy"
Friend_Table:
        Friend_of       = "Cinderella"
        Friend          = "Prince"
Friend_Table:
        Friend_of       = "Snow White"
        Friend          = "Happy"
```

The following tables show a conceptual picture of the Fictional_Characters and Friend_Table object classes, the objects added to the classes, and the link relationship between them:

Fictional_Characters			
Story_Star (char)	Birthday (char)	Age (short)	Friends_of (link)
Cinderella	Once upon a time	19	Cinderella
Snow White	Once upon a time	18	Snow White

```
Retrieved data for 'Story_Star = "Cinderella"
     Cinderella:
          Birthday = Once upon a time
          Age = 19
          Friends_of = Cinderella
          Friend_of = Cinderella
```

There is a direct link between the ″**Friends_of**″ and ″**Friend_of**″ columns of the two tables.

Friend_Table	
Friend_of (char)	Friend (char)
Cinderella	Fairy Godmother
Cinderella	mice

Snow White	Sneezy
Snow White	Sleepy
Cinderella	Prince
Snow White	Happy

Conceptual Picture of a Link Relationship Between Two Object Classes

After the **Fictional_Characters** and **Friend_Table** object classes are created and the objects are added, the retrieved data for `Story_Star = 'Cinderella'` would be:

```
Cinderella:
        Birthday         = Once upon a time
        Age              = 19
        Friends_of       = Cinderella
        Friend_of        = Cinderella
```

To see the expanded relationship between the linked object classes, use the **odmget** command on the **Friend_Table** object class. The retrieved data for the `Friend_of = 'Cinderella'` object class would be:

```
Friend_Table:
        Friend_Of = "Cinderella"
        Friend = "Fairy Godmother"



Friend_Table:
        Friend_of = "Cinderella"
        Friend= "mice"

Friend_Table:
        Friend_of = "Cinderella"
        Friend = "Prince"
```

ODM Method Descriptor

The ODM *method descriptor* gives the definition of an object class with objects that can have associated methods or operations. A method descriptor is a variable defined with the ODM method descriptor type.

The operation or method descriptor value for an object is a character string that can be any command, program, or shell script run by method invocation. A different method or operation can be defined for each object in an object class. The operations themselves are not part of ODM; they are defined and coded by the application programmer.

The method for an object is called by a call to the **odm_run_method** subroutine. The invocation of a method is a synchronous event, causing ODM operation to pause until the operation is completed.

For example, the following code can be input to the ODM create facilities to generate the **Supporting_Cast_Ratings** object class:

```
class Supporting_Cast_Ratings {
        char    Others[20];
        short   Dexterity;
        short   Speed;
        short   Strength;
        method  Do_This;
};
```

In the example, the Do_This descriptor is a method descriptor defined for the **Supporting_Cast_Ratings** object class. The value of the method descriptor can be a string specifying a command, program, or shell script for future invocation by an **odm_run_method** subroutine.

The following code is an example of how to add objects to the Supporting_Cast_Ratings object class:

```
Supporting_Cast_Ratings:
        Others          = "Sleepy"
        Dexterity       = 1
        Speed           = 1
        Strength        = 3
        Do_This   = "echo Sleepy has speed of 1"
Supporting_Cast_Ratings:
        Others          = "Fairy Godmother"
        Dexterity       = 10
        Speed           = 10
        Strength        = 10
        Do_This   = "odmget -q "Others='Fairy Godmother'" Supporting_Cast_Ratings"
```

The **Supporting_Cast_Ratings** table shows a conceptual picture of the Supporting_Cast_Ratings object class with the Do_This method descriptor and operations associated with individual objects in the class.

Supporting_Cast_Ratings				
Others (char)	Dexterity (short)	Speed (short)	Stength (short)	Do_This (method)
Sleepy	1	1	3	echo Sleepy has speed of 1
Fairy Godmother	10	10	10	odmget —q ″Others=’Fairy Godmother’″Supporting_Cast_Ratings″

```
odm_run_method  run of Sleepy's method displays
(using echo):
"Sleepy has speed of 1"
```

Conceptual Picture of an Object Class with a Method Descriptor

After the Supporting_Cast_Ratings object class is created and the objects are added, an invocation (by the **odm_run_method** subroutine) of the method defined for Sleepy would cause the **echo** command to display:

```
Sleepy has speed of 1
```

ODM Object Searches

Many ODM routines require that one or more objects in a specified object class be selected for processing. You can include search criteria in the form of qualifiers when you select objects with certain routines.

qualifier A null-terminated string parameter on ODM subroutine calls that gives the qualification criteria for the objects to retrieve

The descriptor names and qualification criteria specified by this parameter determine which objects in the object class are selected for later processing. Each qualifier contains one or more predicates connected with logical operators. Each predicate consists of a descriptor name, a comparison operator, and a constant.

A qualifier with three predicates joined by two logical operators follows:

```
SUPPNO=30 AND (PARTNO>0 AND PARTNO<101)
```

In this example, the entire string is considered the qualifier. The three predicates are SUPPNO=30, PARTNO>0, and PARTNO<101, and the AND logical operator is used to join the predicates. In the first predicate, SUPPNO is the name of the descriptor in an object, the = (equal sign) is a comparison operator, and 30 is the constant against which the value of the descriptor is compared.

Each predicate specifies a test applied to a descriptor that is defined for each object in the object class. The test is a comparison between the value of the descriptor of an object and the specified constant. The first predicate in the example shows an = (equal to) comparison between the value of a descriptor (SUPPNO) and a constant (30).

The part of the qualifier within parentheses:

```
PARTNO>0 AND PARTNO<101
```

contains two predicates joined by the AND logical operator. The PARTNO descriptor is tested for a value greater than 0 in the first predicate, then tested for a value less than 101 in the second predicate. Then the two predicates are logically concatenated to determine a value for that part of the qualifier. For example, if PARTNO is the descriptor name for a part number in a company inventory, then this part of the qualifier defines a selection for all products with part numbers greater than 0 and less than 101.

In another example, the qualifier:

```
lname='Smith' AND Company.Dept='099' AND Salary<2500
```

can be used to select everyone (in ODM, every object) with a last name of Smith who is in Department 099 and has a salary less than $2500. Note that the Dept descriptor name is qualified with its Company object class to create a unique descriptor.

Descriptor Names in ODM Predicates

In ODM, a descriptor name is not necessarily unique. You can use a descriptor name in more than one object class. When this is the case, you specify the object class name along with the descriptor name in a predicate to create a unique reference to the descriptor.

Comparison Operators in ODM Predicates

The following are valid comparison operators:

=	Equal to
!=	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
LIKE	Similar to; finds patterns in character string data

Comparisons can be made only between compatible data types.

LIKE Comparison Operator

The LIKE operator enables searching for a pattern within a char descriptor type. For example, the predicate:

```
NAME LIKE 'ANNE'
```

defines a search for the value ANNE in the NAME descriptor in each object in the specified object class. In this case, the example is equivalent to:

```
NAME = 'ANNE'
```

You can also use the LIKE operator with the following pattern-matching characters and conventions:

- Use the ? (question mark) to represent any single character. The predicate example:

  ```
  NAME LIKE '?A?'
  ```

  defines a search for any three-character string that has A as a second character in the value of the NAME descriptor of an object. The descriptor values PAM, DAN, and PAT all satisfy this search criterion.
- Use the * (asterisk) to represent any string of zero or more characters. The predicate example:

  ```
  NAME LIKE '*ANNE*'
  ```

  defines a search for any string that contains the value ANNE in the NAME descriptor of an object. The descriptor values LIZANNE, ANNETTE, and ANNE all satisfy this search criterion.
- Use [ ] (brackets) to match any of the characters enclosed within the brackets. The predicate example:

  ```
  NAME LIKE '[ST]*'
  ```

  defines a search for any descriptor value that begins with S or T in the NAME descriptor of an object.

  Use a - (minus sign) to specify a range of characters. The predicate example:

  ```
  NAME LIKE '[AD-GST]*'
  ```

  defines a search for any descriptor value that begins with any of the characters A, D, E, F, G, S, or T.
- Use [!] (brackets enclosing an exclamation mark) to match any single character except one of those enclosed within the brackets. The predicate example:

  ```
  NAME LIKE '[!ST]*'
  ```

  defines a search for any descriptor value except those that begin with S or T in the NAME descriptor of an object.

  You can use the pattern-matching characters and conventions in any combination in the string.

Constants in ODM Predicates

The specified constant can be either a numeric constant or a character string constant.

Numeric Constants in ODM Predicates

Numeric constants in ODM predicates consist of an optional sign followed by a number (with or without a decimal point), optionally followed by an exponent marked by the letter E or e. If used, the letter E or e must be followed by an exponent that can be signed.

Some valid numeric constants are:

```
2            2.545   0.5   -2e5   2.11E0
+4.555e-10   4E0     -10    999   +42
```

The E0 exponent can be used to specify no exponent.

Character String Constants in ODM Predicates

Character string constants must be enclosed in single quotation marks:

```
'smith'   '91'
```

All character string constants are considered to have a variable length. To represent a single quotation mark inside a string constant, use two single quotation marks. For example:

```
'DON''T GO'
```

is interpreted as:

```
DON'T GO
```

AND Logical Operator for Predicates

The AND logical operator can be used with predicates. Use AND or and for the AND logical operator.

The AND logical operator connects two or more predicates. The qualifier example:

```
predicate1 AND predicate2 AND predicate3
```

specifies `predicate1` logically concatenated with `predicate2` followed by the result logically concatenated with `predicate3`.

List of ODM Commands and Subroutines

You can create, add, change, retrieve, display, delete, and remove objects and object classes with ODM. You enter ODM commands on the command line.

You can put ODM subroutines in a C language program to handle objects and object classes. An ODM subroutine returns a value of -1 if the subroutine is unsuccessful. The specific error diagnostic is returned as the **odmerrno** external variable (defined in the **odmi.h** include file). ODM error-diagnostic constants are also included in the **odmi.h** include file.

> **Note:** If the application is linking statically, use option **-binitfini:__odm_initfini_init:__odm_initfini_fini**.

Commands

ODM commands are:

odmadd	Adds objects to an object class. The **odmadd** command takes an ASCII stanza file as input and populates object classes with objects found in the stanza file.
odmchange	Changes specific objects in a specified object class.
odmcreate	Creates empty object classes. The **odmcreate** command takes an ASCII file describing object classes as input and produces C language **.h** and **.c** files to be used by the application accessing objects in those object classes.
odmdelete	Removes objects from an object class.
odmdrop	Removes an entire object class.
odmget	Retrieves objects from object classes and puts the object information into **odmadd** command format.
odmshow	Displays the description of an object class. The **odmshow** command takes an object class name as input and puts the object class information into **odmcreate** command format.

Subroutines

ODM subroutines are:

odm_add_obj	Adds a new object to the object class.
odm_change_obj	Changes the contents of an object.
odm_close_class	Closes an object class.
odm_create_class	Creates an empty object class.
odm_err_msg	Retrieves a message string.
odm_free_list	Frees memory allocated for the **odm_get_list** subroutine.
odm_get_by_id	Retrieves an object by specifying its ID.
odm_get_first	Retrieves the first object that matches the specified criteria in an object class.
odm_get_list	Retrieves a list of objects that match the specified criteria in an object class.
odm_get_next	Retrieves the next object that matches the specified criteria in an object class.

odm_get_obj	Retrieves an object that matches the specified criteria from an object class.
odm_initialize	Initializes an ODM session.
odm_lock	Locks an object class or group of classes.
odm_mount_class	Retrieves the class symbol structure for the specified object class.
odm_open_class	Opens an object class.
odm_rm_by_id	Removes an object by specifying its ID.
odm_rm_obj	Removes all objects that match the specified criteria from the object class.
odm_run_method	Invokes a method for the specified object.
odm_rm_class	Removes an object class.
odm_set_path	Sets the default path for locating object classes.
odm_unlock	Unlocks an object class or group of classes.
odm_terminate	Ends an ODM session.

ODM Example Code and Output

The following Fictional_Characters, Friend_Table, and Enemy_Table Object Classes and Relationships tables list the object classes and objects created by the example code in this section.

Fictional_Characters					
Story_Star (char)	**Birthday (char)**	**Age (short)**	**Friends_of (link)**	**Enemies_of (link)**	**Do_This (method)**
Cinderella	Once upon a time	19	Cinderella	Cinderella	echo Cleans House
Snow White	Once upon a time	18	Snow White	Snow White	echo Cleans House

Friend_Table	
Friend_of (char)	Friend (char)
Cinderella	Fairy Godmother
Cinderella	mice
Snow White	Sneezy
Snow White	Sleepy
Cinderella	Prince
Snow White	Happy

Table 10.

Enemy_Table	
Enemy_of (char)	**Enemy (char)**
Cinderella	midnight
Cinderella	Mean Stepmother
Snow White	Mean Stepmother

ODM Example Input Code for Creating Object Classes

The following example code in the `MyObjects.cre` file creates three object classes when used as an input file with the **odmcreate** command:

```
*      MyObjects.cre
*       An input file for ODM create utilities.
*       Creates three object classes:
*               Friend_Table
*               Enemy_Table
*               Fictional_Characters

class   Friend_Table {
        char    Friend_of[20];
        char    Friend[20];
};

class   Enemy_Table {
        char    Enemy_of[20];
        char    Enemy[20];
};

class   Fictional_Characters {
        char    Story_Star[20];
        char    Birthday[20];
        short   Age;
        link    Friend_Table Friend_Table Friend_of Friends_of;
        link    Enemy_Table Enemy_Table Enemy_of Enemies_of;
        method  Do_This;
};

* End of MyObjects.cre input file for ODM create utilities. *
```

The `Fictional_Characters` object class contains six descriptors:

- `Story_Star` and `Birthday`, each with a descriptor type of char and a 20-character maximum length.
- `Age` with a descriptor type of short.
- Arrange to `Friends_of` and `Enemies_of` are both from the link class, link to the two previously defined object classes.

  Note: Note that the object class link is repeated twice.
- `Do_This` with a descriptor type of method.

The file containing this code must be processed with the **odmcreate** command to generate the object class files required by ODM.

ODM Example Output for Object Class Definitions

Processing the code in the `MyObjects.cre` file with the **odmcreate** command generates the following structures in a **.h** file:

```
* MyObjects.h
* The file output from ODM processing of the MyObjects.cre input
* file. Defines structures for the three object classes:
*       Friend_Table
*       Enemy_Table
*       Fictional_Characters
#include <odmi.h>

struct  Friend_Table {
       long     _id;     * unique object id within object class *
       long     _reserved;  * reserved field *
       long     _scratch;   * extra field for application use *
       char     Friend_of[20];
       char     Friend[20];
};

#define Friend_Table_Descs 2
extern struct Class Friend_Table_CLASS[];
#define get_Friend_Table_list(a,b,c,d,e) (struct Friend_Table * )odm_get_list (a,b,c,d,e)

struct  Enemy_Table {
        long    _id;
        long    _reserved;
```

```
        long    _scratch;
        char    Enemy_of[20];
        char    Enemy[20];
};
#define Enemy_Table_Descs 2
extern struct Class Enemy_Table_CLASS[];
#define get_Enemy_Table_list(a,b,c,d,e) (struct Enemy_Table * )odm_get_list (a,b,c,d,e)

struct  Fictional_Characters {
        long    _id;
        long    _reserved;
        long    _scratch;
        char    Story_Star[20];
        char    Birthday[20];
        short   Age;
       struct  Friend_Table *Friends_of;    * link *
        struct  listinfo *Friends_of_info;   * link *
        char    Friends_of_Lvalue[20];       * link *
        struct  Enemy_Table *Enemies_of;     * link *
        struct  listinfo *Enemies_of_info;   * link *
        char    Enemies_of_Lvalue[20];       * link *
        char    Do_This[256];                * method *
};

#define Fictional_Characters_Descs 6

extern struct Class Fictional_Characters_CLASS[];
#define get_Fictional_Characters_list(a,b,c,d,e) (struct Fictional_Characters * )odm_get_list (a,b,c,d,e)

* End of MyObjects.h structure definition file output from ODM    * processing.
```

ODM Example Code for Adding Objects to Object Classes

The following code can be processed by the **odmadd** command to populate the object classes created by the processing of the `MyObjects.cre` input file:

```
* MyObjects.add
* An input file for ODM add utilities.
* Populates three created object classes:
*       Friend_Table
*       Enemy_Table
*       Fictional_Characters

Fictional_Characters:
Story_Star = "Cinderella" #a comment for the  MyObjects.add file.
Birthday        =       "Once upon a time"
Age             =       19
Friends_of      =       "Cinderella"
Enemies_of      =       "Cinderella"
Do_This =               "echo Cleans house"

Fictional_Characters:
Story_Star      =       "Snow White"
Birthday        =       "Once upon a time"
Age             =       18
Friends_of      =       "Snow White"
Enemies_of      =       "Snow White"
Do_This   =              "echo Cleans house"

Friend_Table:
Friend_of       =       "Cinderella"
Friend          =       "Fairy Godmother"

Friend_Table:
Friend_of       =       "Cinderella"
Friend          =       "mice"

Friend_Table:
Friend_of       =       "Snow White"
Friend          =       "Sneezy"
```

```
Friend_Table:
Friend_of       =       "Snow White"
Friend          =       "Sleepy"

Friend_Table:
Friend_of       =       "Cinderella"
Friend          =       "Prince"

Friend_Table:
Friend_of       =       "Snow White"
Friend          =       "Happy"

Enemy_Table:
Enemy_of        =       "Cinderella"
Enemy           =       "midnight"

Enemy_Table:
Enemy_of        =       "Cinderella"
Enemy           =       "Mean Stepmother"

Enemy_Table:
Enemy_of        =       "Snow White"
Enemy           =       "Mean Stepmother"

* End of MyObjects.add input file for ODM add utilities. *
```

Note: The ***** (asterisk) or the **#** (pound sign) comment above will not go into the object file; it is only for the **.add** file as a comment. The comment will be included in the file and treated as a string if it is included inside the ″ ″ (double quotes).

Chapter 18. sed Program Information

The **sed** program is a text editor that has similar functions to those of **ed**, the line editor. Unlike **ed**, however, the **sed** program performs its editing without interacting with the person requesting the editing.

Manipulating Strings with sed

The **sed** program enables you to to do the following:

- Edit very large files
- Perform complex editing operations many times without requiring extensive retyping and cursor positioning (as interactive editors do)
- Perform global changes in one pass through the input.

The editor keeps only a few lines of the file being edited in memory at one time, and does not use temporary files. Therefore, the file to be edited can be any size as long as there is room for both the input file and the output file in the file system.

Starting the Editor

To use the editor, create a command file containing the editing commands to perform on the input file. The editing commands perform complex operations and require a small amount of typing in the command file. Each command in the command file must be on a separate line. Once the command file is created, enter the following command on the command line:

```
sed -fCommandFile >Output <Input
```

In this command the parameters mean the following:

CommandFile	The name of the file containing editing commands.
Output	The name of the file to contain the edited output.
Input	The name of the file, or files, to be edited.

The **sed** program then makes the changes and writes the changed information to the output file. The contents of the input file are not changed.

How sed Works

The **sed** program is a stream editor that receives its input from standard input, changes that input as directed by commands in a command file, and writes the resulting stream to standard output. If you do not provide a command file and do not use any flags with the **sed** command, the **sed** program copies standard input to standard output without change. Input to the program comes from two sources:

Input stream A stream of ASCII characters either from one or more files or entered directly from the keyboard. This stream is the data to be edited.

Commands A set of addresses and associated commands to be performed, in the following general form:

```
[Line1 [,Line2] ] command [argument]
```

The parameters *Line1* and *Line2* are called addresses. Addresses can be either patterns to match in the input stream, or line numbers in the input stream.

You can also enter editing commands along with the **sed** command by using the **-e** flag.

When **sed** edits, it reads the input stream one line at a time into an area in memory called the pattern space. When a line of data is in the pattern space, **sed** reads the command file and tries to match the addresses in the command file with characters in the pattern space. If it finds an address that matches something in the pattern space, **sed** then performs the command associated with that address on the part of the pattern space that matched the address. The result of that command changes the contents of the pattern space, and thus becomes the input for all following commands.

When **sed** has tried to match all addresses in the command file with the contents of the pattern space, it writes the final contents of the pattern space to standard output. Then it reads a new input line from standard input and starts the process over at the start of the command file.

Some editing commands change the way the process operates.

Flags used with the **sed** command can also change the operation of the command. See “Using the sed Command Summary” for more information.

Using Regular Expressions

A regular expression is a string that contains literal characters, pattern-matching characters and/or operators that define a set of one or more possible strings. The stream editor uses a set of pattern-matching characters that is different from the shell pattern-matching characters, but the same as the line editor, **ed**.

Using the sed Command Summary

All **sed** commands are single letters plus some parameters, such as line numbers or text strings. The commands summarized below make changes to the lines in the pattern space.

The following symbols are used in the syntax diagrams:

Symbol	Meaning
[ ]	Square brackets enclose optional parts of the commands
italics	Parameters in italics represent general names for a name that you enter. For example, *FileName* represents a parameter that you replace with the name of an actual file.
Line1	This symbol is a line number or regular expression to match that defines the starting point for applying the editing command.
Line2	This symbol is a line number or regular expression to match that defines the ending point to stop applying the editing command.

Line Manipulation

Function	Syntax/Description
append lines	[*Line1*]**a\\n***Text* Writes the lines contained in *Text* to the output stream after *Line1*. The **a** command must appear at the end of a line.
change lines	[*Line1* [,*Line2*]]**c\\n***Text* Deletes the lines specified by *Line1* and *Line2* as the *delete lines* command does. Then it writes *Text* to the output stream in place of the deleted lines.

Function	Syntax/Description
delete lines	[*Line1* [,*Line2*]]**d** Removes lines from the input stream and does not copy them to the output stream. The lines not copied begin at line number *Line1*. The next line copied to the output stream is line number *Line2* + 1. If you specify only one line number, then only that line is not copied. If you do not specify a line number, the next line is not copied. You cannot perform any other functions on lines that are not copied to the output.
insert lines	[*Line1*] i **\\n***Text* Writes the lines contained in *Text* to the output stream before *Line1.* The **i** command must appear at the end of a line.
next line	[*Line1* [,*Line2*]]**n** Reads the next line, or group of lines from *Line1* to *Line2* into the pattern space. The current contents of the pattern space are written to the output if it has not been deleted.

Substitution

Function	Syntax/Description
substitution for pattern	[*Line1* [,*Line2*]] **s**/*Pattern/String/Flags* Searches the indicated line(s) for a set of characters that matches the regular expression defined in *Pattern*. When it finds a match, the command replaces that set of characters with the set of characters specified by *String*.

Input and Output

Function	Syntax/Description
print lines	[*Line1* [,*Line2*]] **p** Writes the indicated lines to STDOUT at the point in the editing process that the **p** command occurs.
write lines	[*Line1* [,*Line2*]]**w** *FileName* Writes the indicated lines to a *FileName* at the point in the editing process that the **w** command occurs. If *FileName* exists, it is overwritten; otherwise, it is created. A maximum of 10 different files can be mentioned as input or output files in the entire editing process. Include exactly one space between **w** and *FileName*.
read file	[*Line1*]**r** *FileName* Reads *FileName* and appends the contents after the line indicated by *Line1*. Include exactly one space between **r** and *FileName*. If *FileName* cannot be opened, the command reads it as a null file without giving any indication of an error.

Matching Across Lines

Function	Syntax/Description
join next line	[*Line1* [,*Line2*]]**N** Joins the indicated input lines together, separating them by an embedded new-line character. Pattern matches can extend across the embedded new-lines(s).
delete first line of pattern space	[*Line1* [,*Line2*]]**D** Deletes all text in the pattern space up to and including the first new-line character. If only one line is in the pattern space, it reads another line. Starts the list of editing commands again from the beginning.
print first line of pattern space	[*Line1* [,*Line2*]]**P** Prints all text in the pattern space up to and including the first new-line character to STDOUT.

Pick up and Put down

Function	Syntax/Description
pick up copy	[*Line1* [,*Line2*]]**h** Copies the contents of the pattern space indicated by *Line1* and *Line2* if present, to the holding area.
pick up copy, appended	[*Line1* [,*Line2*]]**H** Copies the contents of the pattern space indicated by *Line1* and *Line2* if present, to the holding area, and appends it to the end of the previous contents of the holding area.
put down copy	[*Line1* [,*Line2*]]**g** Copies the contents of the holding area to the pattern space indicated by *Line1* and *Line2* if present. The previous contents of the pattern space are destroyed.
put down copy, appended	[*Line1* [,*Line2*]]**G** Copies the contents of the holding area to the end of the pattern space indicated by *Line1* and *Line2* if present. The previous contents of the pattern space are not changed. A new-line character separates the previous contents from the appended text.
exchange copies	[*Line1* [,*Line2*]]**x** Exchanges the contents of the holding area with the contents of the pattern space indicated by *Line1* and *Line2* if present.

Control

Function	Syntax/Description
negation	[*Line1* [,*Line2*]]**!** The **!** (exclamation point) applies the command that follows it on the same line to the parts of the input file that are *not* selected by *Line1* and *Line2*.

Function	Syntax/Description

command groups	[*Line1* [,*Line2*]]{
	
	grouped commands
	
	}
	
	The { (left brace) and the } (right brace) enclose a set of commands to be applied as a set to the input lines selected by *Line1* and *Line2*. The first command in the set can be on the same line or on the line following the left brace. The right brace must be on a line by itself. You can nest groups within groups.

labels	:*Label* Marks a place in the stream of editing command to be used as a destination of each branch. The symbol *Label* is a string of up to 8 bytes. Each *Label* in the editing stream must be different from any other *Label*.
branch to label, unconditional	[*Line1* [,*Line2*]]**x***Label* Branches to the point in the editing stream indicated by *Label* and continues processing the current input line with the commands following *Label*. If *Label* is null, branches to the end of the editing stream, which results in reading a new input line and starting the editing stream over. The string *Label* must appear as a *Label* in the editing stream.
test and branch	[*Line1* [,*Line2*]]**t***Label* If any successful substitutions were made on the current input line, branches to *Label*. If no substitutions were made, does nothing. Clears the flag that indicates a substitution was made. This flag is cleared at the start of each new input line.
wait	[*Line1*]**q** Stops editing in an orderly fashion by writing the current line to the output, writing any appended or read test to the output, and stopping the editor.
find line number	[*Line1*]**=** Writes to standard output the line number of the line that matches *Line1*.

Using Text in Commands

The **append**, **insert** and **change** lines commands all use a supplied text string to add to the output stream. This text string conforms to the following rules:

- Can be one or more lines long.
- Each \n (new-line character) inside *Text* must have an additional \ character before it (\\n).
- The *Text* string ends with a new-line that does not have an additional \ character before it (\n).
- Once the command inserts the *Text* string, the string:
  - Is always written to the output stream, regardless of what other commands do to the line that caused it to be inserted.
  - Is not scanned for address matches.

– Is not affected by other editing commands.
– Does not affect the line number counter.

Using String Replacement

The **s** command performs string replacement in the indicated lines in the input file. If the command finds a set of characters in the input file that satisfies the regular expression *Pattern*, it replaces the set of characters with the set of characters specified in *String*.

The *String* parameter is a literal set of characters (digits, letters and symbols). Two special symbols can be used in *String*:

Symbol	Use
&	This symbol in *String* is replaced by the set of characters in the input lines that matched *Pattern*. For example, the command:

```
s/boy/&s/
```

tells **sed** to find a pattern boy in the input line, and copy that pattern to the output with an appended `s`. Therefore, it changes the input line:

From: The boy look at the game.
To: The boys look at the game.

Symbol	Use
\d	**d** is a single digit. This symbol in *String* is replaced by the set of characters in the input lines that matches the **d**th substring in *Pattern*. Substrings begin with the characters \(and end with the characters\ ). For example, the command: `s/\(stu\)\(dy\)/\1r\2/` **From:** The study chair **To:** The sturdy chair

The letters that appear as flags change the replacement as follows:

Symbol	Use
g	Substitutes *String* for all instances of *Pattern* in the indicated line(s). Characters in *String* are not scanned for a match of *Pattern* after they are inserted. For example, the command: `s/r/R/g` changes: **From:** the red round rock **To:** the Red Round Rock
p	Prints (to STDOUT) the line that contains a successfully matched *Pattern*.
w *FileName*	Writes to *FileName* the line that contains a successfully matched *Pattern*. if *FileName* exists, it is overwritten; otherwise, it is created. A maximum of 10 different files can be mentioned as input or output files in the entire editing process. Include exactly one space between **w** and *FileName*.

Chapter 19. Shared Libraries, Shared Memory, and The malloc Subsystem

This chapter provides information about the operating system facilities provided for sharing libraries and memory allocation.

The operating system provides facilities for the creation and use of dynamically bound shared libraries. Dynamic binding allows external symbols referenced in user code and defined in a shared library to be resolved by the loader at run time.

The shared library code is not present in the executable image on disk. Shared code is loaded into memory once in the shared library segment and shared by all processes that reference it. The advantages of shared libraries are:

- Less disk space is used because the shared library code is not included in the executable programs.
- Less memory is used because the shared library code is only loaded once.
- Load time may be reduced because the shared library code may already be in memory.
- Performance may be improved because fewer page faults will be generated when the shared library code is already in memory. However, there is a performance cost in calls to shared library routines of one to eight instructions.

The symbols defined in the shared library code that are to be made available to referencing modules must be explicitly exported using an exports file, unless the -bexpall options is used. The first line of the file optionally contains the path name of the shared library. Subsequent lines contain the symbols to be exported.

Shared Objects and Runtime Linking

By default, programs are linked so that a reference to a symbol imported from a shared object is bound to that definition at load time. This is true even if the program, or another shared object required by the program, defines the same symbol.

Runtime linker A shared object that allows symbols to be rebound for appropriately linked programs

You include the runtime linker in a program by linking the program with the **-brtl** option. This option has the following effects:

- A reference to the runtime linker is added to your program. When program execution begins, the startup code (**/lib/crt0.o**) will call the runtime linker before the main function is called.
- All input files that are shared objects are listed as dependents of your program in your program's loader section. The shared objects are listed in the same order as they were specified on the command line. This causes the system loader to load all these shared objects so that the runtime linker can use their definitions. If the **-brtl** option is not used, a shared object that is not referenced by the program is not listed, even if it provides definitions that might be needed by another shared object used by the program.
- A shared object contained in an archive is only listed if the archive specifies automatic loading for the shared object member. You specify automatic loading for an archive member **foo.o** by creating a file with the following lines:

  ```
  # autoload
  #! (foo.o)
  ```

  and adding the file as a member to the archive.

- In dynamic mode, input files specified with the **-l** flag may end in **.so**, as well as in **.a**. That is, a reference to **-lfoo** is satisfied by the first **libfoo.so** or **libfoo.a** found in any of the directories being searched. Dynamic mode is in effect by default unless the **-bstatic** option is used.

The runtime linker mimics the behavior of the **ld** command when static linking is used, except that only exported symbols can be used to resolve symbols. Even when runtime linking is used, the system loader must be able to load and resolve all symbol references in the main program and any module it depends on. Therefore, if a definition is removed from a module, and the main program has a reference to this definition, the program will not execute, even if another definition for the symbol exists in another module.

The runtime linker can rebind all references to symbols imported from another module. A reference to a symbol defined in the same module as the reference can only be rebound if the module was built with runtime linking enabled for that symbol.

Shared modules shipped with AIX 4.2 or later have runtime linking enabled for most exported variables. Runtime linking for functions is only enabled for functions called through a function pointer. For example, as shipped, calls to the **malloc** subroutine within shared object **shr.o** in **/lib/libc.a** cannot be rebound, even if a definition of **malloc** exists in the main program or another shared module. You can link most shipped shared modules to enable runtime linking for functions as well as variables by running the **rtl_enable** command.

Operation of the Runtime Linker

The main program is loaded and resolved by the system loader in the usual way. If the executable program cannot be loaded for any reason, the **exec()** subroutine fails and the runtime linker is not invoked at all. If the main program loads successfully, control passes to the runtime linker, which rebinds symbols as described below. When the runtime linker completes, initialization routines are called, if appropriate, and then the main function is called.

The runtime linker processes modules in breadth-first search order, starting with the main executable and continuing with the direct dependents of the main executable, according to the order of dependent modules listed in each module's loader section. This order is also used when searching for the defining instance of a symbol. The ″defining instance″ of a symbol is usually the first instance of a symbol, but there are two exceptions. If the first instance of a symbol is an unresolved, deferred import, no defining instance exists. If the first instance is a BSS symbol (that is, with type **XTY_CM**, indicating an uninitialized variable), and there is another instance of the symbol that is neither a BSS symbol nor an unresolved, deferred import, the first such instance of the symbol is the defining instance.

The loader section of each module lists imported symbols, which are usually defined in another specified module, and exported symbols, which are usually defined in the module itself. A symbol that is imported and exported is called a ″passed-through'' import. Such a symbol appears to be defined in one module, although it is actually defined in another module.

Symbols can also be marked as ″deferred imports.″ References to deferred import symbols are never rebound by the runtime linker. Resolution of these symbols must be performed by the system loader, either by calling **loadbind()** or by loading a new module explicitly with **load()** or **dlopen()**.

References to imported symbols (other than deferred imports) can always be rebound. The system loader will have already resolved most imports. References to each imported symbol are rebound to the symbol's defining instance. If no defining instance exists, an error message will be printed to standard error. In addition, if the typechecking hash string of an imported symbol does not match the hash string of the defining symbol, an error message is printed.

References to exported symbols are also rebound to their defining instances, as long as the references appear in the relocation table of the loader section. (Passed-through imports are processed along with other imports, as described above.) Depending on how the module was linked, some references to

exported symbols are bound at link time and cannot be rebound. Since exported symbols are defined in the exporting module, a defining instance of the symbol will always exist, unless the first instance is a deferred import, so errors are unlikely, but still possible, when rebinding exported symbols. As with imports, errors are printed if the typechecking hash strings do not match when a symbol is rebound.

Whenever a symbol is rebound, a dependency is added from the module using the symbol to the module defining the symbol. This dependency prevents modules from being removed from the address space prematurely. This is important when a module loaded by the **dlopen** subroutine defines a symbol that is still being used when an attempt is made to unload the module with the **dlclose** subroutine.

The loader section symbol table does not contain any information about the alignment or length of symbols. Thus, no errors are detected when symbols are rebound to instances that are too short or improperly aligned. Execution errors may occur in this case.

Once all modules have been processed, the runtime linker calls the **exit** subroutine if any runtime linking errors occurred, passing an exit code of `144` (0x90). Otherwise, execution continues by calling initialization routines or **main()**.

Creating a Shared Object with Runtime Linking Enabled

To create a shared object enabled for runtime linking, you link with the **-G** flag. When this flag is used, the following actions take place:

1. Exported symbols are given the `nosymbolic` attribute, so that all references to the symbols can be rebound by the runtime linker.
2. Undefined symbols are permitted (see the **-berok** option). Such symbols are marked as being imported from the symbolic module name `".."`. Symbols imported from `".."` must be resolved by the runtime linker before they can be used because the system loader will not resolve these symbols.
3. The output file is given a module type of `SRE`, as if the **-bM:SRE** option had been specified.
4. All shared objects listed on the command line are listed as dependents of the output module, in the same manner as described when linking a program with the **-brtl** option.
5. Shared objects in archives are listed if they have the `autoload` attribute.

Using the **-berok** option, implied by the **-G** flag, can mask errors that could be detected at link time. If you intend to define all referenced symbols when linking a module, you should use the **-bernotok** option after the **-G** flag. This causes errors to be reported for undefined symbols.

Shared Libraries and Lazy Loading

By default, when a module is loaded, the system loader automatically loads all of the module's dependents at the same time. Loading of dependents occurs because when a module is linked, a list of the module's dependent modules is saved in the loader section of the module.

dump -H Command that allows viewing of dependent modules list.

-blazy In AIX 4.2.1 and later, linker option that links a module so that only some of its dependents are loaded when a function in the module is first used.

When you use lazy loading, you can improve the performance of a program if most of a module's dependents are never actually used. On the other hand, every function call to a lazily loaded module has an additional overhead of about 7 instructions, and the first call to a function requires loading the defining module and modifying the function call. Therefore, if a module calls functions in most of its dependents, lazy loading may not be appropriate.

When a function defined in a lazily loaded module is called for the first time, an attempt is made to load the defining module and find the desired function. If the module cannot be found or if the function is not exported by the module, the default behavior is to print an error message to standard error and exit with a return code of 1. An application can supply its own error handler by calling the function **_lazySetErrorHandler** and supplying the address of an error handler. An error handler is called with 3 arguments: the name of the module, the name of the symbol, and an error value indicating the cause of the error. If the error handler returns, its return value should be the address of a substitute function for the desired function. The return value for **_lazySetErrorHandler** is NULL if no error handler exists, and the address of a previous handler if one exists.

Using lazy loading does not usually change the behavior of a program, but there are a few exceptions. First, any program that relies on the order that modules are loaded is going to be affected, because modules can be loaded in a different order, and some modules might not be loaded at all.

Second, a program that compares function pointers might not work correctly when lazy loading is used, because a single function can have multiple addresses. In particular, if module A calls function ′f’ in module B, and if lazy loading of module B was specified when module A was linked, then the address of ′f’ computed in module A differs from the address of ′f’ computed in other modules. Thus, when you use lazy loading, two function pointers might not be equal, even if they point to the same function.

Third, if any modules are loaded with relative path names and if the program changes working directories, the dependent module might not be found when it needs to be loaded. When you use lazy loading, you should use only absolute path names when referring to dependent modules at link time.

The decision to enable lazy loading is made at link time on a module-by-module basis. In a single program, you can mix modules that use lazy loading with modules that do not. When linking a single module, a reference to a variable in a dependent module prevents that module from being loaded lazily. If all references to a module are to function symbols, the dependent module can be loaded lazily.

The lazy loading facility can be used in both threaded and non-threaded applications.

Lazy Loading Execution Tracing

A runtime feature is provided that allows you to view the loading activity as it takes place. This is accomplished using the environment variable **LDLAZYDEBUG**. The value of this variable is a number, in decimal, octal (leading 0), or hexadecimal (leading 0x) that is the sum of one or more of the following values:

1	Show load or look-up errors. If a required module cannot be found, a message displays and the lazy load error handler is called. If a requested symbol is not available in the loaded referenced module, a message displays before the error handler is called.
2	Write tracing messages to **stderr** instead of **stdout**. By default, these messages are written to the standard output file stream. This value selects the standard error stream.
4	Display the name of the module being loaded. When a new module is required to resolve a function call, the name of the module that is found and loaded displays. This only occurs at the first reference to a function within that module; that is, once a module is loaded, it remains available for subsequent references to functions within that module. Additional load operations are not required.
8	Display the name of the called function. The name of the required function, along with the name of the module from which the function is expected, displays. This information displays before the module is loaded.

Creating a Shared Library

Prerequisite Tasks

1. Create one or more source files that are to be compiled and linked to create a shared library. These files contain the exported symbols that are referenced in other source files.

   For the examples in this article, two source files, `share1.c` and `share2.c`, are used. The `share1.c` file contains the following code:

```
/************
 * share1.c: shared library source.
*************/

#include <stdio.h>

void func1 ()
 {
   printf("func1 called\n");
 }

void func2 ()
 {
   printf("func2 called\n");
 }
```

   The `share2.c` file contains the following code:

```
/************
 * share2.c: shared library source.
*************/

void func3 ()
 {
   printf("func3 called\n");
 }
```

   The exported symbols in these files are `func1`, `func2`, and `func3`.

2. Create a main source file that references the exported symbols that will be contained in the shared library.

   For the examples in this article the main source file named `main.c` is used. The `main.c` file contains the following code:

```
/************
 * main.c: contains references to symbols defined
 * in share1.c and share2.c
*************/

#include <stdio.h>

  extern void func1 (),
                unc2 (),
                func3 ();
main ()
 {
                func1 ();
                func2 ();
                func3 ();
 }
```

3. Create the exports file necessary to explicitly export the symbols in the shared library that are referenced by other object modules.

   For the examples in this article, an exports file named `shrsub.exp` is used. The `shrsub.exp` file contains the following code:

```
#! /home/sharelib/shrsub.o
* Above is full pathname to shared library object file
func1
func2
func3
```

The `#!` line is meaningful only when the file is being used as an import file. In this case, the `#!` line identifies the name of the shared library file to be used at run time.

Procedure

1. Compile and link the two source code files to be shared. (This procedure assumes you are in the **/home/sharedlib** directory.) To compile and link the source files, enter the following commands:

   ```
   cc -c share1.c
   cc -c share2.c
   cc -o shrsub.o share1.o share2.o -bE:shrsub.exp -bM:SRE -bnoentry
   ```

   This creates a shared library name `shrsub.o` in the **/home/sharedlib** directory.

-bM:SRE flag — Marks the resultant object file `shrsub.o` as a re-entrant, shared library

Each process that uses the shared code gets a private copy of the data in its private process area.

flag — Sets the dummy entry point `_nostart` to override the default entry point, `_start`

-bnoentry flag — Tells the linkage editor that the shared library does not have an entry point

A shared library may have an entry point, but the system loader does not make use of an entry point when a shared library is loaded.

2. Use the following command to put the shared library in an archive file:

   ```
   ar qv libsub.a shrsub.o
   ```

   This step is optional. Putting the shared library in an archive makes it easier to specify the shared library when linking your program, because you can use the **-l** and **-L** flags with the **ld** command.

3. Compile and link the main source code with the shared library to create the executable file. (This step assumes your current working directory contains the **main.c** file.) Use the following command:

   ```
   cc -o main main.c -lsub -L/home/sharedlib
   ```

   If the shared library is not in an archive, use the command:

   ```
   cc -o main main.c /home/sharedlib/shrsub.o -L/home/sharedlib
   ```

   The program `main` is now executable. The `func1`, `func2`, and `func3` symbols have been marked for load-time deferred resolution. At run time, the system loader loads the module in to the shared library (unless the module is already loaded) and dynamically resolves the references.

-L flag — Adds the specified directory (in this case, `/home/sharedlib`) to the library search path, which is saved in the loader section of the program.

At run time the library search path is used to tell the loader where to find shared libraries.

LIBPATH environment variable — A colon-separated list of directory paths that can also be used to specify a different library search path. Its format is identical to that of the **PATH** environment variable.

The directories in the list are searched to resolve references to shared objects. The **/usr/lib** and **/lib** directories contain shared libraries and should normally be included in your library search path.

Program Address Space Overview

The Base Operating System provides a number of services for programming application program memory use. Tools are available to assist in allocating memory, mapping memory and files, and profiling application memory usage. As background, this section describes the system's memory management architecture and memory management policy.

System Memory Architecture Introduction

The system employs a memory management scheme that uses software to extend the capabilities of the physical hardware. Because the address space does not correspond one-to-one with real memory, the address space (and the way the system makes it correspond to real memory) is called virtual memory.

The subsystems of the kernel and the hardware that cooperate to translate the virtual address to physical addresses make up the memory management subsystem. The actions the kernel takes to ensure that processes share main memory fairly comprise the memory management policy. The following sections describe the characteristics of the memory management subsystem in greater detail.

The Physical Address Space of 32-bit Systems

The hardware provides a continuous range of virtual memory addresses, from `0x0000000000000` to `0xFFFFFFFFFFFFF`, for accessing data. The total addressable space is more than 1,000 terabytes. Memory access instructions generate an address of 32 bits: 4 bits to select a segment register and 28 bits to give an offset within the segment. This addressing scheme provides access to 16 segments of up to 256M bytes each. Each segment register contains a 24-bit segment ID that becomes a prefix to the 28-bit offset, which together form the virtual memory address. The resulting 52-bit virtual address refers to a single, large, systemwide virtual memory space.

The process space is a 32-bit address space; that is, programs use 32-bit pointers. However, each process or interrupt handler can address only the systemwide virtual memory space (segment) whose segment IDs are in the segment register. A process accesses more than 16 segments by changing registers rapidly.

32-bit processes on 64-bit systems have the same effective address space as on 32-bit systems (2^{32} bytes), but can access the same virtual address space as 64-bit processes (2^{80} bytes).

The Physical Address Space of 64-bit Systems

The hardware provides a continuous range of virtual memory addresses, from `0x00000000000000000000` to `0xFFFFFFFFFFFFFFFFFFFF`, for accessing data. The total addressable space is more than 1 trillion terabytes. Memory access instructions generate an address of 64 bits: 36 bits to select a segment register and 28 bits to give an offset within the segment. This addressing scheme provides access to more than 64 million segments of up to 256M bytes each. Each segment register contains a 52-bit segment ID that becomes a prefix to the 28-bit offset, which together form the virtual memory address. The resulting 80-bit virtual address refers to a single, large, systemwide virtual memory space.

The process space is a 64-bit address space; that is, programs use 64-bit pointers. However, each process or interrupt handler can address only the systemwide virtual memory space (segment) whose segment IDs are in the segment register.

Segment Register Addressing

The system kernel loads some segment registers in the conventional way for all processes, implicitly providing the memory addressability needed by most processes. These registers include two kernel segments, and a shared-library segment, and an I/O device segment, that are shared by all processes and whose contents are read-only to non-kernel programs. There is also a segment for the **exec** system call of a process, which is shared on a read-only basis with other processes executing the same program, a private shared-library data segment that contains read-write library data, and a read-write segment that is private to the process. The remaining segment registers may be loaded using memory mapping techniques to provide more memory, or through memory access to files according to access permissions imposed by the kernel. See “Understanding Memory Mapping” on page 537 for information on the available memory mapping services.

The system’s 32-bit addressing and the access provided through indirection capabilities gives each process an interface that does not depend on the actual size of the systemwide virtual memory space. Some segment registers are shared by all processes, others by a subset of processes, and yet others are accessible to only one process. Sharing is achieved by allowing two or more processes to load the same segment ID.

Paging Space

To accommodate the large virtual memory space with a limited real memory space, the system uses real memory as a work space and keeps inactive data and programs that are not mapped on disk. The area of disk that contains this data is called the paging space. A page is a unit of virtual memory that holds 4K bytes of data and can be transferred between real and auxiliary storage. When the system needs data or a program in the page space, it:

1. Finds an area of memory that is not currently active.
2. Ensures that an up-to-date copy of the data or program from that area of memory is in the paging space on disk.
3. Reads the new program or data from the paging space on disk into the newly freed area of memory.

Memory Management Policy

The real-to-virtual address translation and most other virtual memory facilities are provided to the system transparently by the Virtual Memory Manager (VMM). The VMM implements virtual memory, allowing the creation of segments larger than the physical memory available in the system. It accomplishes this by maintaining a list of free pages of real memory that it uses to retrieve pages that need to be brought into memory.

The VMM occasionally must replenish the pages on the free list by removing some of the current page data from real memory. The process of moving data between memory and disk as the data is needed is called ″paging.″ To accomplish paging, the VMM uses page-stealing algorithms that categorize pages into three classes, each with unique entry and exit criteria:

- working storage pages
- local file pages
- remote file pages

In general, working pages have highest priority, followed by local file pages, and then remote file pages.

In addition, the VMM uses a technique known as the clock algorithm to select pages to be replaced. This technique takes advantage of a referenced bit for each page as an indication of what pages have been recently used (referenced). When a page-stealer routine is called, it cycles through a page frame table, examining each page’s referenced bit. If the page was unreferenced and is stealable (that is, not pinned and meets other page-stealing criteria), it is stolen and placed on the free list. Referenced pages may not

be stolen, but their reference bit is reset, effectively ″aging″ the reference so that the page may be stolen the next time a page-stealing algorithm is issued. See “Paging Space Programming Requirements” on page 564 for more information.

Memory Allocation

Version 3 of the operating system uses a delayed paging slot technique for storage allocated to applications. This means that when storage is allocated to an application with a subroutine such as **malloc**, no paging space is assigned to that storage until the storage is referenced. See ″System Memory Allocation″ (“System Memory Allocation Using the malloc Subsystem” on page 545) to learn more about the system’s allocation policy.

Understanding Memory Mapping

The speed at which application instructions are processed on a system is proportionate to the number of access operations required to obtain data outside of program-addressable memory. The system provides two methods for reducing the transactional overhead associated with these external read and write operations. You can map file data into the process address space. You can also map processes to anonymous memory regions that may be shared by cooperating processes.

Memory mapped files provide a mechanism for a process to access files by directly incorporating file data into the process address space. The use of mapped files can significantly reduce I/O data movement since the file data does not have to be copied into process data buffers, as is done by the **read** and **write** subroutines. When more than one process maps the same file, its contents are shared among them, providing a low-overhead mechanism by which processes can synchronize and communicate.

Mapped memory regions, also called shared memory areas, can serve as a large pool for exchanging data among processes. The available subroutines do not provide locks or access control among the processes. Therefore, processes using shared memory areas must set up a signal or semaphore control method to prevent access conflicts and to keep one process from changing data that another is using. Shared memory areas can be most beneficial when the amount of data to be exchanged between processes is too large to transfer with messages, or when many processes maintain a common large database.

The system provides two methods for mapping files and anonymous memory regions. The following subroutines, known collectively as the **shmat** services, are typically used to create and use shared memory segments from a program:

shmctl	Controls shared memory operations
shmget	Gets or creates a shared memory segment
shmat	Attaches a shared memory segment from a process
shmdt	Detaches a shared memory segment from a process
disclaim	Removes a mapping from a specified address range within a shared memory segment

The **ftok** subroutine provides the key that the **shmget** subroutine uses to create the shared segment

The second set of services, collectively known as the **mmap** services, is typically used for mapping files, although it may be used for creating shared memory segments as well. The mmap services include the following subroutines:

madvise	Advises the system of a process’ expected paging behavior
mincore	Determines residency of memory pages
mmap	Maps an object file into virtual memory
mprotect	Modifies the access protections of memory mapping
msync	Synchronizes a mapped file with its underlying storage device
munmap	Unmaps a mapped memory region

The **msem_init**, **msem_lock**, **msem_unlock**, **msem_remove**, **msleep**, and **mwakeup** subroutines provide access control for the processes mapped using the **mmap** services.

mmap Comparison with shmat

As with the shmat services, the portion of the process address space available for mapping files with the mmap services is dependent on whether a process is a 32-bit process or a 64-bit process. For 32-bit processes, the portion of address space available for mapping consists of addresses in the range of 0x30000000-0xCFFFFFFF, for a total of 2.5G bytes of address space. In AIX 4.2.1 and later, the portion of address space available for mapping files consists of addresses in the rangesof 0x30000000-0xCFFFFFFF and 0x30000000-0xCFFFFFFF, 0xE0000000-0xEFFFFFFF for a total of 2.75G bytes of address space. All available ranges within the 32-bit process address space are available for both fixed-location and variable-location mappings. Fixed-location mappings occur when applications specify that a mapping be placed at a fixed location within the address space. Variable-location mappings occur when applications specify that the system should decide the location at which a mapping should be placed.

For 64-bit processes, two sets of address ranges with the process address space are available for **mmap** or **shmat** mappings. The first, consisting of the single range 0x07000000_00000000-0x07FFFFFF_FFFFFFFF, is available for both fixed-location and variable-location mappings. The second set of address ranges is available for fixed-location mappings only and consists of the ranges 0x30000000-0xCFFFFFFF, 0xE0000000-0xEFFFFFFF, and 0x10_00000000-0x06FFFFFF_FFFFFFFF. The last range of this set, consisting of 0x10_00000000-0x06FFFFFF_FFFFFFFF, is also made available to system loader to hold program text, data and heap, so only unused portions of the range are available for fixed-location mappings.

Both the **mmap** and **shmat** services provide the capability for multiple processes to map the same region of an object such that they share addressability to that object. However, the **mmap** subroutine extends this capability beyond that provided by the **shmat** subroutine by allowing a relatively unlimited number of such mappings to be established. While this capability increases the number of mappings supported per file object or memory segment, it can prove inefficient for applications in which many processes map the same file data into their address space.

The **mmap** subroutine provides a unique object address for each process that maps to an object. The software accomplishes this by providing each process with a unique virtual address, known as an alias. The **shmat** subroutine allows processes to share the addresses of the mapped objects.

Because only one of the existing aliases for a given page in an object has a real address translation at any given time, only one of the **mmap** mappings can make a reference to that page without incurring a page fault. Any reference to the page by a different mapping (and thus a different alias) results in a page fault that causes the existing real-address translation for the page to be invalidated. As a result, a new translation must be established for it under a different alias. Processes share pages by moving them between these different translations.

For applications in which many processes map the same file data into their address space, this toggling process may have an adverse affect on performance. In these cases, the **shmat** subroutine may provide more efficient file-mapping capabilities.

> **Note:** On systems with PowerPC processors, multiple virtual addresses can exist for the same real address. A real address can be aliased to different effective addresses in different processes without toggling. Because there is no toggling, there is no performance degradation.

Use the **shmat** services under the following circumstances:

- For 32-bit application, eleven or fewer files are mapped simultaneously, and each is smaller than 256MB.
- When mapping files larger than 256MB.
- When mapping shared memory regions which need to be shared among unrelated processes (no parent-child relationship).
- When mapping entire files.

Use **mmap** under the following circumstances:

- Portability of the application is a concern.
- Many files are mapped simultaneously.
- Only a portion of a file needs to be mapped.
- Page-level protection needs to be set on the mapping.
- Private mapping is required.

In AIX 4.2.1 and later, an ″extended **shmat**″ capability is available for 32-bit applications with their limited address spaces. If you define the environment variable **EXTSHM=ON**, then processes executing in that environment can create and attach more than eleven shared memory segments. The segments can be from 1 byte to 256M bytes in size. For segments larger than 256M bytes in size, the environment variable **EXTSHM=ON** is ignored. The process can attach these segments into the address space for the size of the segment. Another segment can be attached at the end of the first one in the same 256M byte region. The address at which a process can attach is at page boundaries, which is a multiple of **SHMLBA_EXTSHM** bytes. For segments larger than 256M bytes in size, the address at which a process can attach is at 256M byte boundaries, which is a multiple of **SHMLBA** bytes.

Some restrictions exist on the use of the extended **shmat** feature. These shared memory regions cannot be used as I/O buffers where the unpinning of the buffer occurs in an interrupt handler. The restrictions on the use of extended **shmat** I/O buffers is the same as that of **mmap** buffers.

The environment variable provides the option of executing an application with either the additional functionality of attaching more than 11 segments when **EXTSHM=ON**, or the higher-performance access to 11 or fewer segments when the environment variable is not set. Again, the ″extended **shmat**″ capability only applies to 32-bit processes.

mmap Compatibility Considerations

The mmap services are specified by various standards and commonly used as the file-mapping interface of choice in other operating system implementations. However, the system's implementation of the **mmap** subroutine may differ from other implementations. The **mmap** subroutine incorporates the following modifications:

- Mapping into the process private area is not supported.
- Mappings are not implicitly unmapped. An **mmap** operation which specifies **MAP_FIXED** will fail if a mapping already exists within the range specified.
- For private mappings, the copy-on-write semantic makes a copy of a page on the first write reference.
- Mapping of I/O or device memory is not supported.
- Mapping of character devices or use of an **mmap** region as a buffer for a read-write operation to a character device is not supported.
- The **madvise** subroutine is provided for compatibility only. The system takes no action on the advice specified.
- The **mprotect** subroutine allows the specified region to contain unmapped pages. In operation, the unmapped pages are simply skipped over.

- The OSF/AES-specific options for default exact mapping and for the **MAP_INHERIT**, **MAP_HASSEMAPHORE**, and **MAP_UNALIGNED** flags are not supported.

Using the Semaphore Subroutines

The **msem_init**, **msem_lock**, **msem_unlock**, **msem_remove**, **msleep** and **mwakeup** subroutines conform to the OSF Application Environment specification. They provide an alternative to IPC interfaces such as the **semget** and **semop** subroutines. Benefits of using the semaphores include an efficient serialization method and the reduced overhead of not having to make a system call in cases where there is no contention for the semaphore.

Semaphores should be located in a shared memory region. Semaphores are specified by **msemaphore** structures. All of the values in a **msemaphore** structure should result from a **msem_init** subroutine call. This call may or may not be followed by a sequence of calls to the **msem_lock** subroutine or the **msem_unlock** subroutine. If a **msemaphore** structure values originated in another manner, the results of the semaphore subroutines are undefined.

The address of the **msemaphore** structure is significant. You should be careful not to modify the structure's address. If the structure contains values copied from a **msemaphore** structure at another address, the results of the semaphore subroutines are undefined.

The semaphore subroutines may prove less efficient when the semaphore structures exist in anonymous memory regions created with the **mmap** subroutine, particularly in cases where many processes reference the same semaphores. In these instances, the semaphore structures should be allocated out of shared memory regions created with the **shmget** and **shmat** subroutines.

Mapping Files with the shmat Subroutine

Mapping can be used to reduce the overhead involved in writing and reading the contents of files. Once the contents of a file are mapped to an area of user memory, the file may be manipulated as if it were data in memory, using pointers to that data instead of input/output calls. The copy of the file on disk also serves as the paging area for that file, saving paging space.

A program can use any regular file as a mapped data file. You can also extend the features of mapped data files to files containing compiled and executable object code. Because mapped files can be accessed more quickly than regular files, the system can load a program more quickly if its executable object file is mapped to a file.See “Creating a Mapped Data File with the shmat Subroutine” on page 543 for information on using any regular file as a mapped data file.

To create a program as a mapped executable file, compile and link the program using the **-K** flag with the **cc** or **ld** command. The **-K** flag tells the linker to create an object file with a page-aligned format. That is, each part of the object file starts on a page boundary (an address that can be divided by 2K bytes with no remainder). This option results in some empty space in the object file but allows the executable file to be mapped into memory. When the system maps an object file into memory, the text and data portions are handled differently.

Copy-on-Write Mapped Files

To prevent changes made to mapped files from appearing immediately in the file on disk, map the file as a copy-on-write file. This option creates a mapped file with changes that are saved in the system paging space, instead of to the copy of the file on disk. You must choose to write those changes to the copy on disk to save the changes. Otherwise, you lose the changes when closing the file.

Because the changes are not immediately reflected in the copy of the file that other users may access, use copy-on-write mapped files only among processes that cooperate with each other.

The system does not detect the end of files mapped with the **shmat** subroutine. Therefore, if a program writes beyond the current end of file in a copy-on-write mapped file by storing into the corresponding memory segment (where the file is mapped), the actual file on disk is extended with blocks of zeros in preparation for the new data. If the program does not use the **fsync** subroutine before closing the file, the data written beyond the previous end of file is not written to disk. The file appears larger, but contains only the added zeros. Therefore, always use the **fsync** subroutine before closing a copy-on-write mapped file to preserve any added or changed data. See "Creating a Copy-On-Write Mapped Data File with the shmat Subroutine" on page 544 for additional information.

Mapping Shared Memory Segments with the shmat Subroutine

The system uses shared memory segments similarly to the way it creates and uses files. Defining the terms used for shared memory with respect to the more familiar file-system terms is critical to understanding shared memory. A definition list of shared memory terms follows:

Term	Definition
key	The unique identifier of a particular shared segment. It is associated with the shared segment as long as the shared segment exists. In this respect, it is similar to the *file name* of a file.
shmid	The identifier assigned to the shared segment for use within a particular process. It is similar in use to a *file descriptor* for a file.
attach	Specifies that a process must attach a shared segment in order to use it. Attaching a shared segment is similar to opening a file.
detach	Specifies that a process must detach a shared segment once it is finished using it. Detaching a shared segment is similar to closing a file.

See "Creating a Shared Memory Segment with the shmat Subroutine" on page 544 for additional information.

Related Information

"Program Address Space Overview" on page 535.

"Creating a Mapped Data File with the shmat Subroutine" on page 543.

"Creating a Copy-On-Write Mapped Data File with the shmat Subroutine" on page 544.

IPC (Inter-Process Communication) Limits

This document describes how to set limits for IPC mechanisms and applies to AIX 3.2.5, AIX 4.1, AIX 4.2, and AIX 4.3.

Shared Memory Segments

On some UNIX systems, users edit **/etc/master** and set their own limits for IPC mechanisms (semaphore, shared memory segments, and message queues). The problem with this method is that the higher the limits are set, the bigger the kernel gets, and performance can be adversely affected. AIX uses a different method.

In AIX, upper limits are set for IPC mechanisms, and the individual IPC types are dynamically allocated/deallocated up to these upper limits. These are not configurable in AIX.

Therefore, the kernel grows and shrinks in size as IPC types are allocated, so any performance hit is only for the life of the IPC type.

This difference in methods sometimes confuses users who are installing or using databases. In AIX, IPC limits are handled for users. The limit that may cause a problem is the maximum number of shared

memory segments per process. This number can be 10, 11 or more if EXTSHM is used. In other words the limit that may cause a problem is the maximum number of shared memory regions that can be attached simultaneously per process.

The structures containing IPC limits are defined in three files in **/usr/include/sys/**: **sem.h**, **msg.h**, and **shm.h**. The structures themselves are called `seminfo`, `msginfo`, and `shminfo`, respectively. Only the structures are defined—not the contents.

The following is a list of values for AIX 3.2.5, 4.1, 4.2, 4.3.0 4.3.1, 4.3.2 and later. None of these values can be modified.

Before AIX 4.2.1

- In these versions, a single shared memory region, whatever its size, always consumes a 256MB region of the address space.
- Only 10 regions can be attached to a process.

AIX 4.2.1

- AIX 4.2.1 provides enhancements to the number of shared memory regions a process can attach. A process can attach to 11 shared memory regions, each up to 256MB in size.
- AIX 4.2.1 also provides the ability to attach more than 10 shared memory regions to a process when the process is created in a shell with an environment variable defined (for example, `EXTSHM=ON`). In this environment, a shared memory region can be as small as one page in size (4096 bytes) and as large as 256MB. The address space consumed is exactly the size of the shared memory region. The number of regions a process can attach is now limited only by the available address space. The total amount of address space available in this mode is also 11*256MB.
- The only change in any values is the maximum bytes on queue, which has changed from 64KB to 4MB.

AIX 4.3

- There were no limit changes in AIX 4.3.

AIX 4.3.1

- Provides all the features of AIX 4.3. The only change in any values is the maximum size of a shared memory segment which has changed from 256MB to 2GB.

AIX 4.3.2

- Provides all the features of AIX 4.3.1. The only change in any values is the maximum number of message queues, semaphore sets and shared memory segments which has changed from 4096 to 131072.
- AIX 4.3.2 included a change to increase the maximum number of messages per queue from 8192 to 524288.

AIX VERSIONS	3.2.5 - 4.2.0	4.2.1	4.3.0	4.3.1	4.3.2
Semaphores:					
Maximum number of semaphore IDs	4096	4096	4096	4096	131072
Maximum semaphores per semaphore ID	65535	65535	65535	65535	65535
Maximum operations per semop call	1024	1024	1024	1024	1024
Maximum undo entries per process	1024	1024	1024	1024	1024
Size in bytes of undo structure	8208	8208	8208	8208	8208
Semaphore maximum value	32767	32767	32767	32767	32767
Adjust on exit maximum value	16384	16384	16384	16384	16384
Message Queues:					
Maximum message size	65535	4MB	4MB	4MB	4MB
Maximum bytes on queue	65535	4MB	4MB	4MB	4MB
Maximum number of message queue IDs	4096	4096	4096	4096	131072
Maximum messages per queue ID	8192	524288	524288	524288	524288
Shared Memory:					
Maximum segment size	256MB	256MB	256MB	2GB	2GB
Minimum segment size	1	1	1	1	1
Maximum number of shared memory IDs.	4096	4096	4096	4096	131072
Maximum number of segments per process	10	11*	11*	11*	11*

***** See the information in preceding sections of this document about the differences between the various versions.

Creating a Mapped Data File with the shmat Subroutine

Prerequisite Condition

The file to be mapped is a regular file.

Procedure

The creation of a mapped data file is a two-step process. First, you create the mapped file. Then, because the **shmat** subroutine does not provide for it, you must program a method for detecting the end of the mapped file.

1. To create the mapped data file:
   a. Open (or create) the file and save the file descriptor:

```
if( ( fildes = open( filename , 2 ) ) < 0 )
{
        printf( "cannot open file\n" );
        exit(1);
}
```

   b. Map the file to a segment with the **shmat** subroutine:

```
file_ptr=shmat (fildes, 0, SHM_MAP);
```

      The SHM_MAP constant is defined in the **/usr/include/sys/shm.h** file. This constant indicates that the file is a mapped file. Include this file and the other shared memory header files in a program with the following directives:

```
#include <sys/shm.h>
```

2. To detect the end of the mapped file:
   a. Use the **lseek** subroutine to go to the end of file:

```
eof = file_ptr + lseek(fildes, 0, 2);
```

This example sets the value of `eof` to an address that is 1 byte beyond the end of file. Use this value as the end-of-file marker in the program.

b. Use `file_ptr` as a pointer to the start of the data file, and access the data as if it were in memory:

```
while ( file_ptr < eof)
{
      .
      .
      .
      (references to file using file_ptr)
}
```

Note: The **read** and **write** subroutines also work on mapped files and produce the same data as when pointers are used to access the data.

c. Close the file when the program is finished working with it:

```
close (fildes );
```

Creating a Copy-On-Write Mapped Data File with the shmat Subroutine

Prerequisite Condition

The file to be mapped is a regular file.

Procedure

1. Open (or create) the file and save the file descriptor:

```
if( ( fildes = open( filename , 2 ) ) < 0 )
{
        printf( "cannot open file\n" );
        exit(1);
}
```

2. Map the file to a segment as copy-on-write, with the **shmat** subroutine:

```
file_ptr = shmat( fildes, 0, SHM_COPY );
```

   The `SHM_COPY` constant is defined in the **/usr/include/sys/shm.h** file. This constant indicates that the file is a copy-on-write mapped file. Include this header file and other shared memory header files in a program with the following directives:

```
#include <sys/shm.h>
```

3. Use `file_ptr` as a pointer to the start of the data file, and access the data as if it were in memory.

```
while ( file_ptr < eof)
{
      .
      .
      .
      (references to file using file_ptr)
}
```

4. Use the **fsync** subroutine to write changes to the copy of the file on disk to save the changes:

```
fsync( fildes );
```

5. Close the file when the program is finished working with it:

```
close( fildes );
```

Creating a Shared Memory Segment with the shmat Subroutine

Prerequisite Tasks or Conditions

None.

Procedure

1. Create a key to uniquely identify the shared segment. Use the **ftok** subroutine to create the key. For example, to create the key `mykey` using a project ID of `R` contained in the variable `proj` (type **char**) and a file name of `null_file`, use a statement like:

   ```
   mykey = ftok( null_file, proj );
   ```

2. Either:
   - Create a shared memory segment with the **shmget** subroutine. For example, to create a shared segment that contains 4096 bytes and assign the **shmid** to an integer variable `mem_id`, use a statement like:

     ```
     mem_id = shmget(mykey, 4096, IPC_CREAT | o666 );
     ```

   - Get a previously created shared segment with the **shmget** subroutine. For example, to get a shared segment that is already associated with the key `mykey` and assign the **shmid** to an integer variable `mem_id`, use a statement like:

     ```
     mem_id = shmget( mykey, 4096, IPC_ACCESS );
     ```

3. Attach the shared segment to the process with the **shmat** subroutine. For example, to attach a previously created segment, use a statement like:

   ```
   ptr = shmat( mem_id );
   ```

   In this example, the variable `ptr` is a pointer to a structure that defines the fields in the shared segment. Use this template structure to store and retrieve data in the shared segment. This template should be the same for all processes using the segment.
4. Work with the data in the segment using the template structure.
5. Detach from the segment using the **shmdt** subroutine:

   ```
   shmdt( ptr );
   ```

6. If the shared segment is no longer needed, remove it from the system with the **shmctl** subroutine:

   ```
   shmctl( mem_id, IPC_RMID, ptr );
   ```

   Note: You can also use the **ipcs** command to get information about a segment, and the **ipcrm** command to remove a segment.

System Memory Allocation Using the malloc Subsystem

Memory is allocated to applications using the malloc subsystem. The malloc subsystem is a memory management API that consists of the following subroutines:

- **malloc**
- **calloc**
- **realloc**
- **free**
- **mallopt**
- **mallinfo**
- **alloca**
- **valloc**

The malloc subsystem manages a logical memory object called a heap. The heap is a region of memory that resides in the application's address space between the last byte of data allocated by the compiler and the end of the data region. The heap is the memory object from which memory is allocated and to which memory is returned by the malloc subsystem API.

The malloc subsystem performs three fundamental memory operations: allocation, deallocation, and reallocation. Allocation is performed by the **malloc** and **calloc** subroutines, deallocation by the **free**

subroutine, and reallocation by the **realloc** subroutine. The **mallopt** and **mallinfo** subroutines are supported for System V compatibility. The **mallinfo** subroutine can be used during program development to obtain information about the heap managed by the **malloc** subroutine. The **mallopt** subroutine can be used to disclaim page-aligned, page-sized free memory, and to enable and disable the default allocator. Similar to the **malloc** subroutine, the **valloc** subroutine is provided for Berkeley compatibility.

Refer to the following sections for additional information:

- Working with the Heap
- “Understanding System Allocation Policy” on page 547
- “Understanding the Default Allocation Policy” on page 547
- “Understanding the 3.1 Allocation Policy” on page 548
- “Comparison of the Default and 3.1 Allocation Policies” on page 550

Working with the Heap

A 32-bit application program running on the system has an address space that is divided into seven segments, as follows:

`0x00000000` to `0x0fffffff`	Contains the kernel.
`0x10000000` to `0x1fffffff`	Contains the application program text.
`0x20000000` to `0x2fffffff`	Contains the application program data and the application stack.
`0x30000000` to `0xafffffff`	Available for use by shared memory or **mmap** services.
`0xd0000000` to `0xdfffffff`	Contains shared library text.
`0xe0000000` to `0xefffffff`	Contains miscellaneous kernel data.
`0xf0000000` to `0x0fffffff`	Contains the application shared library data.

Working with the Heap

A 64-bit application program running on the system has an address space that is divided into seven segments, as follows:

`0x0000 0000 0000 0000` to `0x0000 0000 0fff ffff`	Contains the kernel.
`0x0000 0000 d000 0000` to `0x0000 0000 dfff ffff`	Contains shared library information.
`0x0000 0000 e000 0000` to `0x0000 0000 efff ffff`	Contains miscellaneous kernel data.
`0x0000 0000 f000 0000` to `0x0000 0000 0fff ffff`	Reserved.
`0x0000 0001 0000 0000` to `0x07ff ffff ffff ffff`	Contains the application program text and application program data and the application stack and shared memory or **mmap** services.
`0x0800 0000 0000 0000` to `0x08ff ffff ffff ffff`	Privately loaded objects.
`0x0900 0000 0000 0000` to `0x09ff ffff ffff ffff`	Shared library text and data.
`0x0f00 0000 0000 0000` to `0x0fff ffff ffff ffff`	Application stack.

The _edata location is an identifier that points to the first byte following the last byte of program data. The heap is created by the malloc subsystem when the first block of data is allocated. The **malloc** subroutine creates the heap by calling the **sbrk** subroutine to move the _edata location up to make room for the heap. The **malloc** subroutine then expands the heap as the needs of the application dictate. Space for the heap is acquired in increments determined by the **BRKINCR** value. This value can be examined with the **mallinfo** subroutine.

The heap is divided into allocated blocks and freed blocks. The free pool consists of the memory available for subsequent allocation. An allocation is completed by first removing a block from the free pool and then returning to the free pool a pointer to this block. A reallocation is completed by allocating a block of storage of the new size, moving the data to the new block, and freeing the original block. The allocated blocks consist of the pieces of the heap being used by the application. Because the memory blocks are

not physically removed from the heap (they simply change state from free to in-use), the size of the heap does not decrease when memory is freed by the application.

Understanding System Allocation Policy

The allocation policy refers to the set of data structures and algorithms employed to represent the heap and to implement allocation, deallocation, and reallocation. The malloc subsystem supports two allocation policies: the default allocation policy and the 3.1 allocation policy. The interface to the malloc subsystem is identical for both allocation policies.

The default allocation policy is generally more efficient and is the preferred choice for the majority of applications. The 3.1 allocation policy has some unique behavioral characteristics that may be beneficial in specific circumstances, as described under “Comparison of the Default and 3.1 Allocation Policies” on page 550. However, the 3.1 allocation policy is only available for use with 32-bit applications. It is not supported for 64-bit applications.

Understanding the Default Allocation Policy

The default allocation policy maintains the free space in the heap as a free tree. The free tree is a binary tree in which nodes are sorted vertically by length and horizontally by address. The data structure imposes no limitation on the number of block sizes supported by the tree, allowing a wide range of potential block sizes. Tree reorganization techniques optimize access times for node location, insertion, and deletion, and also protect against fragmentation.

The default allocation policy provides support for the following optional capabilities:

- “Malloc Multiheap” on page 559
- “Malloc Buckets” on page 560
- “Debug Malloc” on page 554

Allocation

The number of bytes required for a block is calculated using a roundup function. The equation is:

```
If x mod y = 0, then
Roundup(x,y) = x
otherwise,
Roundup(x,y) = (x/y rounded down to the nearest whole number + 1)y

p = sizeof(prefix)=8

pad = Roundup(len + p,16)
```

The leftmost node of the tree that is greater than or equal to the size of the **malloc** subroutine *len* parameter value is removed from the tree. If the block found is larger than the needed size, the block is divided into two blocks: one of the needed size, and the second a remainder. The second block, called the runt, is returned to the free tree for future allocation. The first block is returned to the caller.

If a block of sufficient size is not found in the free tree, the heap is expanded, a block the size of the acquired extension is added to the free tree and allocation continues as previously described.

Deallocation

Memory blocks deallocated with the **free** subroutine are returned to the tree, at the root. Each node along the path to the insertion point for the new node is examined to see if it adjoins the node being inserted. If it does, the two nodes are merged and the newly merged node is relocated in the tree. Length determines the depth of a node in the tree. If no neighbor is found, the node is simply inserted at the appropriate place in the tree. Merging adjacent blocks can significantly reduce heap fragmentation.

Reallocation

If the size of the reallocated block will be larger than the original block, the original block is returned to the free tree with the **free** subroutine so that any possible coalescence can occur. Then, a new block of the requested size is allocated, the data is moved from the original block to the new block, and the new block is returned to the caller.

If the size of the reallocated block is smaller than the original block, the block is split and the runt is returned to the free tree.

Understanding the 3.1 Allocation Policy

The 3.1 allocation policy can be invoked by entering:

```
MALLOCTYPE=3.1; export MALLOCTYPE
```

Thereafter, all 32-bit programs run by the shell will use the 3.1 allocation policy (64-bit programs will continue to use the default allocation policy). Setting MALLOCTYPE to anything other than 3.1 causes the default allocation policy to be used.

The 3.1 allocation policy maintains the heap as a set of 28 hash buckets, each of which points to a linked list. Hashing is a method of transforming a search key into an address for the purpose of storing and retrieving items of data. The method is designed to minimize average search time. A bucket is one or more fields in which the result of an operation is kept. Each linked list contains blocks of a particular size. The index into the hash buckets indicates the size of the blocks in the linked list. The size of the block is calculated using the following formula:

$$size = 2^{i+4}$$

where `i` identifies the bucket. This means that the blocks in the list anchored by bucket zero are $2^{0+4} = 16$ bytes long. Therefore, given that a prefix is 8 bytes in size, these blocks can satisfy requests for blocks between 0 and 8 bytes long. The following table illustrates how requested sizes are distributed among the buckets.

> **Note:** This algorithm can use as much as twice the amount of memory actually allocated by the application. An extra page is required for buckets larger than 4096 bytes because objects of a page in size or larger are page-aligned. Since the prefix immediately precedes the block, an entire page is required solely for the prefix.

3.1 Allocation Policy			
Bucket	**Block Size**	**Sizes Mapped**	**Pages Used**
0	16	0 ... 8	
1	32	9 ... 24	
2	64	25 ... 56	
3	128	57 ... 120	
4	256	121 ... 248	
5	512	249 ... 504	
6	1K	505 ... 1K-8	
7	2K	1K-7 ... 2K-8	
8	4K	2K-7 ... 4K-8	2
9	8K	4K-7 ... 8K-8	3
10	16K	8K-7 ... 16K-8	5
11	32K	16K-7 ... 32K-8	9
12	64K	32K-7 ... 64K-8	17

13	128K	64K-7 ... 128K-8	33
14	256K	128K-7 ... 256K-8	65
15	512K	256K-7 ... 512K-8	129
16	1M	256K-7 ... 1M-8	257
17	2M	1M-7 ... 2M-8	513
18	4M	2M-7 ... 4M-8	1K + 1
19	8M	4M-7 ... 8M-8	2K + 1
20	16M	8M-7 ... 16M-8	4K + 1
21	32M	16M-7 ... 32M-8	8K + 1
22	64M	32M-7 ... 64M-8	16K + 1
23	128M	64M-7 ... 128M-8	32K + 1
24	256M	128M-7 ... 256M-8	64K + 1
25	512M	256M-7 ... 512M-8	128K + 1
26	1024M	512M-7 ... 1024M-8	256K + 1
27	2048M	1024M-7 ... 2048M-8	512K + 1

Allocation

A block is allocated from the free pool by first converting the requested bytes to an index in the bucket array, using the following equation:

```
needed = requested + 8

If needed <= 16,
then
bucket = 0

If needed > 16,
then
bucket = (log(needed)/log(2) rounded down to the nearest whole number) - 3
```

The size of each block in the list anchored by the bucket is $block\ size = 2^{bucket + 4}$. If the list in the bucket is null, memory is allocated using the **sbrk** subroutine to add blocks to the list. If the block size is less than a page, then a page is allocated using the **sbrk** subroutine, and the number of blocks arrived at by dividing the block size into the page size are added to the list. If the block size is equal to or greater than a page, needed memory is allocated using the **sbrk** subroutine, and a single block is added to the free list for the bucket. If the free list is not empty, the block at the head of the list is returned to the caller. The next block on the list then becomes the new head.

Deallocation

When a block of memory is returned to the free pool, the bucket index is calculated as with allocation. The block to be freed is then added to the head of the free list for the bucket.

Reallocation

When a block of memory is reallocated, the needed size is compared against the existing size of the block. Because of the wide variance in sizes handled by a single bucket, the new block size often maps to the same bucket as the original block size. In these cases, the length of the prefix is updated to reflect the new size and the same block is returned. If the needed size is greater than the existing block, the block is freed, a new block is allocated from the new bucket, and the data is moved from the old block to the new block.

Limitations

The 3.1 allocation policy is available for use with 32-bit applications only. If MALLOCTYPE=3.1 is specified for a 64-bit application, the default allocation policy will be used instead.

The 3.1 allocation policy does not support any of the following capabilities:

- Malloc Multiheap
- Malloc Buckets
- Debug Malloc

Comparison of the Default and 3.1 Allocation Policies

The 3.1 allocation policy has been widely used in UNIX systems. However, because it rounds up the size of each allocation request to the next power of 2, it can produce considerable virtual- and real-memory fragmentation and poor locality of reference. The default allocation policy is generally a better choice because it allocates exactly the amount of space requested and is more efficient about reclaiming previously used blocks of memory.

Unfortunately, some application programs may depend inadvertently on side effects of the 3.1 allocation policy for acceptable performance or even for correct functioning. For example, a program that overruns the end of an array may function correctly when using the 3.1 allocator only because of the additional space provided by the rounding-up process. The same program is likely to experience erratic behavior or even fail when used with default allocator because the default allocator only allocates the number of bytes requested.

As another example, because of the inefficient space reclamation of the 3.1 allocation algorithm, the application program almost always receives space that has been set to zeros (when a process touches a given page in its working segment for the first time, that page is set to zeros). Applications may depend on this side effect for correct execution. In fact, zeroing out of the allocated space is not a specified function of **malloc** and would result in an unnecessary performance penalty for programs that initialize only as required and possibly not to zeros. Because the default allocator is more aggressive about reusing space, programs that are dependent on receiving zeroed storage from **malloc** will probably fail when the default allocator is used.

Similarly, if a program continually **realloc**s a structure to a slightly greater size, the 3.1 allocator may not need to move the structure very often. In many cases, **realloc** can make use of the extra space provided by the rounding implicit in the 3.1 allocation algorithm. The default allocator will usually have to move the structure to a slightly larger area because of the likelihood that something else has been **malloc**ed just above it. This may present the appearance of a degradation in **realloc** performance when the default allocator is used instead of the 3.1 allocator. In reality, it is the surfacing of a cost that is implicit in the application program's structure.

User Defined Malloc Replacement

The AIX memory subsystem (malloc, calloc, realloc, free, mallopt and mallinfo) has been modified to allow users to replace it with one of their own design.

> **NOTE:** Replacement Memory Subsystems written in C++ are not supported due to the use of the libc.a memory subsystem in the C++ library libC.a.

The existing memory subsystem works for both threaded and non-threaded applications. The user defined memory subsystem should be thread-safe so that it works in both threaded and non-threaded processes, however, there are no checks to verify that it is. So, if a non-thread safe memory module is loaded in a threaded application, memory and data may be corrupted.

The user defined memory subsystem 32 and 64 bit objects must be placed in an archive with the 32bit shared object named `mem32.o` and the 64bit shared object named `mem64.o`.

The user shared objects must export the following symbols :

- __malloc__

- __free__
- __realloc__
- __calloc__
- __mallinfo__
- __mallopt__
- __malloc_init__
- __malloc_prefork_lock__
- __malloc_postfork_unlock__

The functions are defined as follows:

void *__malloc__(size_t) :
This function is the user equivalent of `malloc`() as described in the AIX documentation.

void __free__(void *) :
This function is the user equivalent of `free`() as described in the AIX documentation.

void *__realloc__(void *, size_t) :
This function is the user equivalent of `realloc`() as described in the AIX documentation.

void *__calloc__(size_t, size_t) :
This function is the user equivalent of `calloc`() as described in the AIX documentation.

int __mallopt__(int, int) :
This function is the user equivalent of `mallopt`() as described in the AIX documentation.

struct mallinfo __mallinfo__() :
This function is the user equivalent of `mallinfo`() as described in the AIX documentation.

The following interfaces are used by the thread subsystem to manage the user defined memory subsystem in a multi-threaded environment. The are only called if the application and/or the user defined module are bound with libpthreads.a. Even if the the user defined subsystem is not thread-safe and not bound with libpthreads.a these symbols must be defined and exported or the object will not be loaded.

void __malloc_init__(void)
Called by the pthread initialization routine. This function is used to initialize the threaded user memory subsystem. In most cases, this includes creating and initializing some form of locking data. Even if the user defined memory subsystem module is bound with libpthreads.a the user defined memory subsystem MUST work before `__malloc_init__()` is called.

void __malloc_prefork_lock__(void)
Called by pthreads when `fork()` is called. This function is used to insure that the memory subsystem is in a known state before the `fork()` and stays that way until the `fork()` has returned. In most cases this includes acquiring the memory subsystem locks.

void __malloc_postfork_unlock__(void)
Called by pthreads when `fork()` is called. This function is used to make the memory subsystem available in the parent and child after a `fork()`. This should undo the work done by `__malloc_prefork_lock__()`. In most cases this includes releasing the memory subsystem locks.

All of the functions must be exported from a shared module. There must be separate modules for 32 and 64 bit implementations place in an archive. For example:

- **mem.exp**:

```
__malloc__
__free__
__realloc__
__calloc__
__mallopt__
```

```
__mallinfo__
__malloc_init__
__malloc_prefork_lock__
__malloc_postfork_unlock__
```

- **mem_functions32.o**:

  Contains all of the required 32 bit functions
- **mem_functions64.o**:

  Contains all of the required 64 bit functions
- Creating 32bit shared object:

```
ld -b32 -m -o mem32.o mem_functions32.o \
-bE:mem.exp \
-bM:SRE -lpthreads -lc
```

- Creating 64bit shared object:

```
ld -b64 -m -o mem64.o mem_functions64.o \
-bE:mem.exp \
-bM:SRE -lpthreads -lc
```

- Creating the archive (the shared objects name must be `mem32.o` for the 32bit object and `mem64.o` for the 64bit object):

```
ar -X32_64 -r archive_name mem32.o mem64.o
```

> **NOTE:** In the above examples for creating the shared objects `-lpthreads` is only needed if the object uses pthread functions.

Enablement

The user defined memory subsystem can be enabled by using either:

- the `MALLOCTYPE` environment variable, or
- the global variable `_malloc_user_defined_name` in the user's application

To use the `MALLOCTYPE` environment variable, the archive containing the user defined memory subsystem is specified by setting `MALLOCTYPE` to `user:`*`archive_name`* where *archive_name* is in the application's `libpath` or the path is specified in the `LIBPATH` environment variable.

To use the global variable`_malloc_user_defined_name`, the user's application must declare the global variable as:

```
char *_malloc_user_defined_name="archive_name"
```

where *archive_name* must be in the application's libpath or a path specified in the `LIBPATH` environment variable.

> **NOTES:**
>
> 1. When a setuid application is exec'd, the `LIBPATH` environment variable is ignored so the archive must be in the application's libpath.
> 2. *archive_name* cannot contain path information.
> 3. When both the environment variable, `MALLOCTYPE`, and the global variable, `_malloc_user_defined_name`, are used to specify the *archive_name*, the archive specified by `MALLOCTYPE` will override the one specified by `_malloc_user_defined_name`.

32/64bit Considerations

If the archive does not contain both the 32 and 64 bit shared objects and the user defined memory subsystem was enabled using the `MALLOCTYPE` environment variable, there will be problems exec'ing 64bit processes from 32bit applications and 32bit processes from 64bit applications. When an new process is created using exec(), it inherits the environment of the calling application. This means that `MALLOCTYPE` will

be inherited and the new process will attempt to load the user defined memory subsystem. If the archive member does not exist for this type of executable, the load will fail and the new process will exit.

Thread Considerations

All of the provided memory functions must work in a multi-threaded environment. Even if the module is linked with libpthreads.a, at least __malloc__() MUST work before __malloc_init__() is called and pthreads is initialized. This is required because the pthread initialization requires malloc() before __malloc_init__() is called.

All provided memory functions must work in both threaded and non-threaded environments. The __malloc__() function should be able to run to completion without having any dependencies on __malloc_init__() (i.e. __malloc__() should initially assume that __malloc_init__() has NOT yes run.) Once __malloc_init__() has completed, then __malloc__() can rely on any work done by __malloc_init__(). This is required because the pthread initialization uses malloc() before __malloc_init__() is called.

There are two variables provided to keep from calling thread related routines when they are not needed. The first, __multi_threaded, is zero until a thread is created when it becomes non-zero and for that process will not be reset to zero. The other variable, __n_pthreads, is -1 until pthreads has been initialized when it is set to 1. From that point on it is a count of the number of active threads.

Example:
If __malloc__() uses pthread_mutex_lock() the code may look something like this:

```
if (__multi_threaded)
pthread_mutex_lock(mutexptr);

/* ..... work ....... */

if (__multi_threaded)
pthread_mutex_unlock(mutexptr);
```

Not only does this keep __malloc__() from executing pthread functions before pthreads is fully initialized, it also speeds up single threaded applications because locking is not done until a second thread is started.

Limitations

Memory subsystems written in C++ are not supported due to initialization and the dependencies of libC.a and the libc.a memory subsystem.

Error messages are not translated due to setlocale() using malloc() to initialize the locales. If malloc() fails then setlocale() cannot finish and the application is still in the POSIX locale so only the default English messages will be displayed.

Existing statically built executables will not be able to use the user defined memory subsystem without recompiling.

Error Reporting

The first time malloc() is called, the 32 or 64 bit object in the archive specified by the MALLOCTYPE environment variable is loaded. If the load fails, a message will be displayed and the application will exit. If the load is successful, an attempt is made to verify that all of the required symbols are present. If any symbols are missing the application will be terminated and the list of missing symbols will displayed.

Related Information

"Creating a Shared Library" on page 533

"Chapter 11. Threads Programming Guidelines" on page 215

The **malloc**, **free**, **realloc**, **calloc**, **mallopt**, **mallinfo**, **alloca**, or **valloc** subroutine.

Debug Malloc

Debugging applications that are mismanaging memory allocated via the malloc() subroutine can be difficult and tedious. Most often, the problem is that data is written past the end of an allocated buffer. Since this has no immediate consequence, problems don't become apparent until much later when the space that was overwritten (usually belonging to another allocation) is used and no longer contains the data originally stored there.

The AIX memory subsystem includes an optional debug capability to allow users to identify memory overwrites, overreads, duplicate frees and reuse of freed memory allocated by malloc(). Activation and configuration of the Debug Malloc capability is available at process startup via the **MALLOCTYPE** and **MALLOCDEBUG** environment variables.

Memory problems detected by Debug Malloc result in an abort() or a segmentation violation **(SIGSEGV)**. In most cases, when an error is detected the application stops immediately and a core file is produced.

Debug Malloc is only available for applications using the default allocator. It is not supported for the AIX 3.1 malloc.

Enabling Debug Malloc

Debug Malloc is not enabled by default. It is enabled and configured by setting the following environment variables:

```
MALLOCTYPE

MALLOCDEBUG
```

To enable Debug Malloc with default settings, set the **MALLOCTYPE** environment variable as follows:

```
MALLOCTYPE=debug
```

To enable Debug Malloc with user-specified configuration options, set both the MALLOCTYPE and MALLOCDEBUG environment variables as follows:

```
MALLOCTYPE=debug

MALLOCDEBUG=options
```

where options is a comma-separated list of one or more predefined configuration options, as described in the next section of this document.

If the application being debugged calls malloc() frequently, it may be necessary to enable the application to access additional memory via use of the the ulimit command and the -bmaxdata option of the ld command. Please refer to the section entitled ″Disk and Memory Considerations″ later in this document for additional information.

MALLOCDEBUG Options

The **MALLOCDEBUG** environment variable can be used to provide Debug Malloc with one or more of the following predefined configuration options:

- align:n
- postfree_checking
- validate_ptrs
- override_signal_handling
- allow_overreading

- report_allocations
- record_allocations

Each of these options is described in detail later in this document.

The **MALLOCDEBUG** environment variable is set using the following syntax:

```
MALLOCDEBUG=[[ align:n | postfree_checking | validate_ptrs |
               override_signal_handling | allow_overreading |
               report_allocations | record_allocations],...]
```

More than one option can be specified (and in any order) as long as options are comma-separated, as in the following example:

```
MALLOCDEBUG=align:0,validate_ptrs,report_allocations
```

Each configuration option should only be specified once when setting **MALLOCDEBUG**. If a configuration option is specified more than once per setting, only the final instance will apply.

It is important to remember that the MALLOCDEBUG environment variable will only be recognized by the malloc subsystem if MALLOCTYPE is set to ″debug″, as in the following example:

```
MALLOCTYPE=debug

MALLOCDEBUG=align:2,postfree_checking,override_signal_handling
```

Each of the **MALLOCDEBUG** options is described in detail below:

align:n By default, malloc() returns a pointer aligned on a 2-word boundary (4-word in 64bit mode). The Debug Malloc align:n option can be used to change the default alignment, where n is the number of bytes to be aligned and can be any power of 2 between 0 and 4096 inclusive (e.g. 0, 1, 2, 4, ...). The values 0 and 1 are treated as the same, i.e., there is no alignment so any memory accesses outside the allocated area will cause an abort().

NOTES:

1. Please refer to the section entitled ″Additional Information about align:n Option″ later in this document for additional information about the align:n option.
2. For allocated space to be word aligned, specify align:n with a value of 4.
3. Applications built using DCE components are restricted to a value of 8 for the align:n option. Values other than 8 may result in undefined behavior.

postfree_checking

By default, the malloc subsystem allows the calling program to access memory that has previously been freed. This is, of course, an error in the calling program. If the Debug Malloc postfree_checking option is specified, any attempt to access memory after it is freed will cause Debug Malloc to report the error and abort the program. A core file will be produced.

NOTE:

1. Specifying the postfree_checking option automatically enables the validate_ptrs option.

validate_ptrs

By default, free() does not validate its input pointer to ensure that it actually references memory previously allocated by malloc(). If the parameter passed to free() is a NULL value, free() will return to the caller without taking any action. If the parameter is invalid, the results will be undefined. A core dump may or may not occur in this case, depending upon the value of the invalid parameter. Specifying the Debug Malloc validate_ptrs option will cause free() to perform

extensive validation on its input parameter. If the parameter is found to be invalid (i.e., it does not reference memory previously allocated by a call to malloc() or realloc()), Debug Malloc will print an error message stating why it is invalid. The abort() function is then called to terminate the process and produce a core file.

override_signal_handling

Debug Malloc reports errors in one of two ways:

1. Memory access errors (such as trying to read or write past the end of allocated memory) will cause a segmentation violation (SIGSEGV), resulting in a core dump.
2. For other types of errors (such as trying to free space that was already freed), Debug Malloc will output an error message, then call abort(), which will send a SIGIOT signal to end the current process.

If the calling program is blocking or catching the SIGSEGV and/or the SIGIOT signals, Debug Malloc will be prevented from reporting errors. The Debug Malloc override_signal_handling option provides a means of addressing this situation without recoding and rebuilding the application.

If the Debug Malloc override_signal_handling option is specified, Debug Malloc will perform the following actions upon each call to one of the memory allocation routines (malloc(), free(), realloc() or calloc()):

1. Disable any existing signal handlers set up by the application.
2. Set the action for both `SIGIOT` and `SIGSEGV` to the default (`SIG_DFL`).
3. Unblock both `SIGIOT` and `SIGSEGV`.

If an application signal handler modifies the action for SIGSEGV between memory allocation routine calls and then attempts an invalid memory access, Debug Malloc will be unable to report the error (the application will not exit and no core file will be produced).

NOTES:

1. The override_signal_handling option may be ineffective in a threaded application environment because Debug Malloc uses sigprocmask() and many threaded processes use pthread_sigmask().
2. If a thread calls sigwait() without including `SIGSEGV` and `SIGIOT` in the signal set and Debug Malloc subsequently detects an error, the thread will hang because Debug Malloc can only generate `SIGSEGV` or `SIGIOT`.
3. If a pointer to invalid memory is passed to a kernel routine, the kernel routine will fail and usually return with errno set to `EFAULT`. If the application is not checking the return from the system call, this error can go undetected.

allow_overreading

By default, Debug Malloc will respond with a segmentation violation and a core dump if the calling program attempts to read past the end of allocated memory. Specifying the Debug Malloc allow_overreading option will cause Debug Malloc to ignore ″overreads″ of this nature so that other types of errors, which may be considered more serious, can be detected first.

report_allocations

Specifying the Debug Malloc report_allocations option will cause Debug Malloc to report all active allocation records at application exit. An active allocation record will be listed for any memory allocation that was not freed prior to application exit. Each allocation record will contain the information listed below under the description for the record_allocations option.

NOTES:

1. Specifying the report_allocations option automatically enables the record_allocations option.
2. One allocation record will always be listed for the atexit() handler that dumps the allocation records.

record_allocations

Specifying the Debug Malloc record_allocations option will cause Debug Malloc to create an allocation record for each malloc() request. Each record contains the following information:

- the original address returned to the caller from malloc().
- a six function traceback starting from the call to malloc().

Each allocation record will be retained until the memory associated with it is freed.

Additional Information about align:n Option

The following formula can be used to calculate how many bytes of overreads and/or overwrites Debug Malloc will allow for a given allocation request when MALLOCDEBUG=align:n and size is the number of bytes to be allocated:

```
((((size / n) + 1) * n) - size) % n
```

The following examples demonstrate the effect of the align:n option on the application's ability to perform overreads and/or overwrites with Debug Malloc enabled:

1. In the example below, the align:n option is specified with a value of 2:

   ```
   MALLOCTYPE=debug
   MALLOCDEBUG=align:2,postfree_checking,override_signal_handling
   ```

   In this case, Debug Malloc will handle overreads and overwrites as follows:

   - When an even number of bytes is allocated, Debug Malloc will allocate exactly the number of bytes requested, which will allow for 0 bytes of overreads and/or overwrites.
   - When an odd number of bytes is allocated, Debug Malloc will allocate the number of bytes requested, plus one additional byte to satisfy the required alignment. This will allow for 1 byte of overreads and/or overwrites.
2. In the example below, the align:n option is specified with a value of 0:

   ```
   MALLOCTYPE=debug
   MALLOCDEBUG=align:0,postfree_checking,override_signal_handling
   ```

   In this case, Debug Malloc will allow 0 bytes of overreads and/or overwrites in all cases, regardless of the number of bytes requested.

Debug Malloc Output

All memory problems detected by Debug Malloc result in an abort() or a segmentation violation (SIGSEGV). If Debug Malloc is enabled and the application runs to completion without an abort() or a segmentation violation, then the malloc subsystem did not detect any memory problems.

In most cases, when an error is detected the application stops immediately and a core file is produced. If the error is caused by an attempt to read or write past the end of allocated memory or to access freed memory, then a segmentation violation will occur at the instruction accessing the memory. If a memory routine (malloc(), free(), realloc() or calloc()) detects an error, a message is displayed and abort() is called.

Performance Considerations

Because of the extra work involved in making various run-time checks, malloc subsystem performance will degrade considerably with Debug Malloc enabled, but not to the point that applications will become unusable. Because of this performance degradation, applications should only be be run with Debug Malloc enabled when trying to debug a known problem. Once the problem is resolved, Debug Malloc should be turned off to restore malloc subsystem performance.

Disk and Memory Considerations

With Debug Malloc enabled, the malloc subsystem will consume significantly more memory. Each malloc() request is increased by ″4096+2*sizeof(unsigned long)″ and then rounded up to the next multiple of PAGESIZE. Debug Malloc may prove to be too memory-intensive to use for some large applications, but for the majority of applications that need memory debugging, the extra use of memory should not cause a problem.

If the application being debugged calls malloc() frequently, it may encounter memory usage problems with Debug Malloc enabled which may prevent the application from executing properly in a single segment. If this occurs, it may be helpful to enable the application to access additional memory via use of the the ulimit command and the -bmaxdata option of the ld command.

For the purpose of running with Debug Malloc enabled, the ulimit should be set for both data (-d) and stack (-s) as follows:

```
ulimit -d unlimited
ulimit -s unlimited
```

The -bmaxdata option should be specified as -bmaxdata:0x80000000 in order to reserve the maximum of 8 segments for a 32-bit process.

When Debug Malloc is turned off, the default values for ulimit and -bmaxdata should be restored.

The **ulimit** command and the **-bmaxdata** option are discussed in detail in “Chapter 8. Large Program Support” on page 157.

Limitations

Debug Malloc is only available for applications using the default allocator. It is not supported for the AIX 3.1 malloc.

Debug Malloc is not appropriate for full-time, constant or system-wide use. Although it is designed for minimal performance impact upon the application being debugged, it may have significant negative impact upon overall system throughput if it is used widely throughout a system. In particular, setting `MALLOCTYPE=debug` in the /etc/environment file (to enable Debug Malloc for the entire system) is unsupported, and will likely cause significant system problems such as excessive use of paging space. Debug Malloc should only be used to debug single applications or small groups of applications at the same time.

In addition, please note that Debug Malloc is not appropriate for use in some debugging situations. Because Debug Malloc places each individual memory allocation on a separate page, programs that issue many small allocation requests will see their memory usage increase dramatically. These programs may encounter new failures as memory allocation requests are denied due to a lack of memory or paging space. These failures are not necessarily errors in the program being debugged, and they are not errors in Debug Malloc.

One specific example of this is the X server, which issues numerous tiny allocation requests during its initialization and operation. Any attempt to run the X server (via the X or xinit commands) with Debug Malloc enabled will result in the failure of the X server due to a lack of available memory. This is a known

limitation of the Debug Malloc tool. However, X clients in general will not encounter functional problems running under Debug Malloc. To use Debug Malloc on an X client program, take the following steps:

1. Start the X server with Debug Malloc turned off.
2. Start a terminal window (e.g. dtterm, xterm, aixterm).
3. Set the appropriate environment variables within the terminal window session to turn Debug Malloc on.
4. Invoke the X client program to be debugged from within the same window.

Related Information

"Chapter 8. Large Program Support" on page 157

malloc, free, realloc, calloc, mallopt, mallinfo, alloca, or valloc Subroutine in *AIX 5L Version 5.1 Technical Reference: Base Operating System and Extensions Volume 1*.

Malloc Multiheap

By default, the malloc subsystem uses a single heap, or free memory pool. However, it also provides an optional multiheap capability to allow users to enable the use of multiple heaps of free memory, rather than just one.

The purpose of providing multiple heap capability in the malloc subsystem is to improve the performance of threaded applications running on multiprocessor systems. When the malloc subsystem is limited to using a single heap, simultaneous memory allocation requests received from threads running on separate processors are serialized, meaning that the malloc subsystem can only service one thread at a time. This can have a serious impact on multiprocessor system performance.

With malloc multiheap capability enabled, the malloc subsystem creates a fixed number of heaps for its use. It will begin to use multiple heaps after the second thread is started (process becomes multithreaded). Each memory allocation request will be serviced using one of the available heaps. The malloc subsystem can then process memory allocation requests in parallel, as long as the number of threads simultaneously requesting service is less than or equal to the number of heaps.

If the number of threads simultaneously requesting service exceeds the number of heaps, then additional simultaneous requests will be serialized. Unless this occurs on an ongoing basis, the overall performance of the malloc subsystem should be significantly improved when multiple threads are making calls to **malloc()** in a multiprocessor environment.

Activation and configuration of the malloc multiheap capability is available at process startup via the **MALLOCMULTIHEAP** environment variable. The maximum number of heaps available with malloc multiheap enabled is 32.

Enabling Malloc Multiheap

Malloc multiheap is not enabled by default. It is enabled and configured by setting the **MALLOCMULTIHEAP** environment variable.

To enable malloc multiheap with default settings, set the **MALLOCMULTIHEAP** environment variable to any non-null value, as follows:

```
MALLOCMULTIHEAP=true
```

Setting **MALLOCMULTIHEAP** in this manner will enable malloc multiheap in its default configuration, with all 32 heaps and the fast heap selection algorithm.

To enable malloc multiheap with user-specified configuration options, set the **MALLOCMULTIHEAP** environment variable as follows:

```
MALLOCMULTIHEAP=options
```

where options is a comma-separated list of one or more predefined configuration options, as described in the next section of this document.

MALLOCMULTIHEAP Options

MALLOCMULTIHEAP environment variable options are as follows:

- heaps:n
- considersize

Each of these options is described in detail later in this document.

The **MALLOCMULTIHEAP** environment variable is set using the following syntax:

```
MALLOCMULTIHEAP=[heaps:n] | [considersize]
```

One or both options can be specified in any order as long as options are comma-separated, as in the following example:

```
MALLOCMULTIHEAP=heaps:3,considersize
```

In the above example, malloc multiheap would be enabled with three heaps and a somewhat slower heap selection algorithm that tries to minimize process size.

Each configuration option should only be specified once when setting **MALLOCMULTIHEAP**. If a configuration option is specified more than once per setting, only the final instance will apply.

Each of the **MALLOCMULTIHEAP** options is described below:

heaps:n
: By default, the maximum number of heaps available to malloc multiheap is 32. The *heaps:n* option can be used to change the maximum number of heaps to any value from 1 through 32, where *n* is the number of heaps. If *n* is set to a value outside the given range, the default value of 32 is used.

considersize
: By default, malloc multiheap selects the next available heap. If the considersize option is specified, malloc multiheap will use an alternate heap selection algorithm that tries to select an available heap that has enough free space to handle the request. This may minimize the working set size of the process by reducing the number of **sbrk()** calls. However, because of the additional processing required, the *considersize* heap selection algorithm is somewhat slower than the default heap selection algorithm.

Malloc Buckets

Malloc buckets provides an optional buckets-based extension of the default allocator. It is intended to improve malloc performance for applications that issue large numbers of small allocation requests. When malloc buckets is enabled, allocation requests that fall within a predefined range of block sizes are processed by malloc buckets. All other requests are processed in the usual manner by the default allocator.

Malloc buckets is not enabled by default. It is enabled and configured prior to process startup by setting the MALLOCTYPE and MALLOCBUCKETS environment variables.

Bucket Composition and Sizing

A bucket consists of a block of memory that is subdivided into a predetermined number of smaller blocks of uniform size, each of which is an allocatable unit of memory. Each bucket is identified using a bucket number. The first bucket is bucket 0, the second bucket is bucket 1, the third bucket is bucket 2, and so on. The first bucket is the smallest and each bucket after that is larger in size than the preceding bucket, using a formula described later in this section. A maximum of 128 buckets is available per heap.

The block size for each bucket is a multiple of a bucket sizing factor. The bucket sizing factor equals the block size of the first bucket. Each block in the second bucket is twice this size, each block in the third bucket is three times this size, and so on. Therefore, a given bucket's block size is determined as follows:

```
block size = (bucket number + 1) * bucket sizing factor
```

For example, a bucket sizing factor of 16 would result in a block size of 16 bytes for the first bucket (bucket 0), 32 bytes for the second bucket (bucket 1), 48 bytes for the third bucket (bucket 2), and so on.

The bucket sizing factor must be a multiple of 8 for 32-bit implementations and a multiple of 16 for 64-bit implementations in order to guarantee that addresses returned from malloc subsystem functions are properly aligned for all data types.

The bucket size for a given bucket is determined as follows:

```
bucket size = number of blocks per bucket * (malloc overhead +
              ((bucket number + 1) * bucket sizing factor))
```

The above formula can be used to determine the actual number of bytes required for each bucket. In this formula, malloc overhead refers to the size of an internal malloc construct that is required for each block in the bucket. This internal construct is 8 bytes long for 32-bit applications and 16 bytes long for 64-bit applications. It is not part of the allocatable space available to the user, but is part of the total size of each bucket.

Number of blocks per bucket, number of buckets and bucket sizing factor are all configurable by setting the MALLOCBUCKETS environment variable.

Processing Allocations from the Buckets

A block will be allocated from one of the buckets whenever malloc buckets is enabled and an allocation request falls within the range of block sizes defined by the buckets. Each allocation request is serviced from the smallest possible bucket to conserve space.

If an allocation request is received for a bucket and all of its blocks are already allocated, malloc buckets will automatically enlarge the bucket to service the request. The number of new blocks added to enlarge a bucket is always equal to the number of blocks initially contained in the bucket, which is configurable by setting the MALLOCBUCKETS environment variable.

Support for Multiheap Processing

The malloc multiheap capability provides a means to enable multiple malloc heaps to improve the performance of threaded applications running on multiprocessor systems. Malloc buckets supports up to 128 buckets per heap. This allows the malloc subsystem to support concurrent enablement of malloc buckets and malloc multiheap so that threaded processes running on multiprocessor systems can benefit from the buckets algorithm.

Enabling Malloc Buckets

Malloc buckets is not enabled by default. It is enabled and configured by setting the following environment variables:

- MALLOCTYPE
- MALLOCBUCKETS

To enable malloc buckets with default settings, set the MALLOCTYPE environment variable as follows:

```
MALLOCTYPE=buckets
```

To enable malloc buckets with user-specified configuration options, set both the MALLOCTYPE and MALLOCBUCKETS environment variables as follows:

```
MALLOCTYPE=buckets
MALLOCBUCKETS=options
```

Where, *options* is a comma-separated list of one or more predefined configuration options, as defined in the next section of this document, Malloc Buckets Configuration Options.

Note: The following malloc subsystem capabilities are mutually exclusive.

- 3.1 Malloc (MALLOCTYPE=3.1)
- Debug Malloc (MALLOCTYPE=debug)
- User Defined Malloc (MALLOCTYPE=user:*archive_name*)
- Malloc Buckets (MALLOCTYPE=buckets)

Malloc Buckets Configuration Options

The MALLOCBUCKETS environment variable can be used to provide malloc buckets with one or more of the following predefined configuration options:

```
number_of_buckets:n
bucket_sizing_factor:n
blocks_per_bucket:n
bucket_statistics:[stdout|stderr|pathname]
```

Each of these options is described in detail in "MALLOCBUCKETS Options".

The MALLOCBUCKETS environment variable is set using the following syntax:

```
MALLOCBUCKETS=[[ number_of_buckets:n | bucket_sizing_factor:n | blocks_per_bucket:n |
bucket_statistics:[stdout|stderr|pathname]],...]
```

More than one option can be specified (and in any order) as long as options are comma-separated, for example:

```
MALLOCBUCKETS=number_of_buckets:128,bucket_sizing_factor:8,bucket_statistics:stderr
MALLOCBUCKETS=bucket_statistics:stdout,blocks_per_bucket:512
```

Commas are the only delimiters that are valid for separating configuration options in this syntax. The use of other delimiters (such as blanks) between options will cause configuration options to be parsed incorrectly.

Each configuration option should only be specified once when setting MALLOCBUCKETS. If a configuration option is specified more than once per setting, only the final instance will apply.

If a configuration option is specified with an invalid value, malloc buckets will write a warning message to standard error and then continue execution using a documented default value.

It is important to remember that the MALLOCBUCKETS environment variable will only be recognized by the malloc subsystem if MALLOCTYPE is set to buckets, as in the following example:

```
MALLOCTYPE=buckets
MALLOCBUCKETS=number_of_buckets:8,bucket_statistics:stderr
```

MALLOCBUCKETS Options

number_of_buckets:*n*

The number_of_buckets:*n* option can be used to specify the number of buckets available per heap, where *n* is the number of buckets. The value specified for *n* will apply to all available heaps.

The default value for number_of_buckets is 16. The minimum value allowed is 1. The maximum value allowed is 128.

bucket_sizing_factor:*n*

The bucket_sizing_factor:*n* option can be used to specify the bucket sizing factor, where *n* is the bucket sizing factor in bytes.

The value specified for bucket_sizing_factor must be a multiple of 8 for 32-bit implementations and a multiple of 16 for 64-bit implementations. The default value for bucket_sizing_factor is 32 for 32-bit implementations and 64 for 64-bit implementations.

blocks_per_bucket:*n*

The blocks_per_bucket:*n* option can be used to specify the number of blocks initially contained in each bucket, where *n* is the number of blocks. This value is applied to all of the buckets. The value of *n* is also used to determine how many blocks to add when a bucket is automatically enlarged because all of its blocks have been allocated.

The default value for blocks_per_bucket is 1024.

bucket_statistics:[stdout|stderr|*pathname***]**

The bucket_statistics option will cause the malloc subsystem to output a statistical summary for malloc buckets upon normal termination of each process that calls the malloc subsystem while malloc buckets is enabled. This summary will show buckets configuration information and the number of allocation requests processed for each bucket. If multiple heaps have been enabled by way of malloc multiheap, the number of allocation requests shown for each bucket will be the sum of all allocation requests processed for that bucket for all heaps.

The buckets statistical summary will be written to one of the following output destinations, as specified with the bucket_statistics option.

- **stdout** - standard output
- **stderr** - standard error
- *pathname* - a user-specified pathname

If a user-specified pathname is provided, statistical output will be appended to the existing contents of the file (if any).

Standard output should not be used as the output destination for a process whose output is piped as input into another process.

The bucket_statistics option is disabled by default.

Notes:

1. One additional allocation request will always be shown in the first bucket for the atexit() handler that prints the statistical summary.
2. For threaded processes, additional allocation requests will be shown for some of the buckets due to malloc subsystem calls issued by the pthreads library.

Malloc Buckets Default Configuration

The following table summarizes the malloc buckets default configuration.

Configuration Option	Default Value (32-bit)	Default Value (64-bit)
number of buckets per heap	16	16
bucket sizing factor	32 bytes	64 bytes
allocation range	1 to 512 bytes (inclusive)	1 to 1024 bytes (inclusive)
number of blocks initially contained in each bucket	1024	1024
bucket statistical summary	disabled	disabled

The default configuration for malloc buckets should be sufficient to provide a performance improvement for many applications that issue large numbers of small allocation requests. However, it may be possible to achieve additional gains by setting the MALLOCBUCKETS environment variable to modify the default configuration. Developers who wish to modify the default configuration should first become familiar with the application's memory requirements and usage. Malloc buckets can then be enabled with the bucket_statistics option to fine tune the buckets configuration.

Limitations

Malloc buckets is only available for applications using the default allocator. It is not supported for the AIX 3.1 malloc.

Because of variations in memory requirements and usage, some applications may not benefit from the memory allocation scheme used by malloc buckets. Therefore, it is not advisable to enable malloc buckets system-wide. For optimal performance, malloc buckets should be enabled and configured on a per-application basis.

Paging Space Programming Requirements

The amount of paging space required by an application depends on the type of activities performed on the system. If paging space runs low, processes may be lost. If paging space runs out, the system may panic. When a paging space low condition is detected, additional paging space should be defined.

The system monitors the number of free paging space blocks and detects when a paging space shortage exists. The **vmstat** command obtains statistics related to this condition. When the number of free paging space blocks falls below a threshold known as the paging space warning level, the system informs all processes (excepts **kprocs**) of the low condition by sending the **SIGDANGER** signal.

> **Note:** If the shortage continues and falls below a second threshold known as the paging space kill level, the system sends the **SIGKILL** signal to processes that are the major users of paging space and that do not have a signal handler for the **SIGDANGER** signal (the default action for the **SIGDANGER** signal is to ignore the signal). The system continues sending **SIGKILL** signals until the number of free paging space blocks is above the paging space kill level.

Processes that dynamically allocate memory can ensure that sufficient paging space exists by monitoring the paging space levels with the **psdanger** subroutine or by using special allocation routines. Processes can avoid being ended when the paging space kill level is reached by defining a signal handler for the **SIGDANGER** signal and by using the **disclaim** subroutine to release memory and paging space resources allocated in the data and stack areas, and in shared memory segments.

Other subroutines that can assist in dynamically retrieving paging information from the VMM include the following:

mincore	Determines the residency of memory pages.
madvise	Permits a process to advise the system about its expected paging behavior.
swapqry	Returns paging device status.
swapon	Activates paging or swapping to a designated block device.

List of Memory Manipulation Services

The memory functions operate on arrays of characters in memory called memory areas. These subroutines enable you to:

- Locate a character within a memory area
- Copy characters between memory areas

- Compare contents of memory areas
- Set a memory area to a value.

You do not need to specify any special flag to the compiler in order to use the memory functions. However, you must include the header file for these functions in your program. To include the header file, use the following statement:

```
#include <memory.h>
```

The following memory services are provided:

compare_and_swap	Compares and swaps data
fetch_and_add	Updates a single word variable atomically
fetch_and_and or **fetch_and_or**	Set or clear bits in a single word variable atomically
malloc, **free**, **realloc**, **calloc**, **mallopt**, **mallinfo**, or **alloca**	Allocate memory
memccpy, **memchr**, **memcmp**, **memcpy**, **memset or memmove**	Perform memory operations.

moncontrol	Starts and stops execution profiling after initialization by the **monitor** subroutine
monitor	Starts and stops execution profiling using data areas defined in the function parameters
monstartup	Starts and stops execution profiling using default-sized data areas
msem_init	Initializes a semaphore in a mapped file or shared memory region
msem_lock	Locks a semaphore
msem_remove	Removes a semaphore
msem_unlock	Unlocks a semaphore
msleep	Puts a process to sleep when a semaphore is busy
mwakeup	Wakes up a process that is waiting on a semaphore
disclaim	Disclaims the content of a memory address range
ftok	Generates a standard interprocess communication key
getpagesize	Gets the system page size
psdanger	Defines the amount of free paging space available
shmat	Attaches a shared memory segment or a mapped file to the current process
shmctl	Controls shared memory operations
shmdt	Detaches a shared memory segment
shmget	Gets a shared memory segment
swapon	Activates paging or swapping to a designated block device
swapqry	Returns device status

List of Memory Mapping Services

The memory mapping subroutines operate on memory regions that have been mapped with the **mmap** subroutine. These subroutines enable you to:

- Map an object file into virtual memory
- Synchronize a mapped file
- Determine residency of memory pages
- Determine access protections to a mapped memory region
- Unmap mapped memory regions.

You do not need to specify any special flag to the compiler to use the memory functions. However, you must include the header file for some of these subroutines. If the subroutine description specifies a header file, you can include it with the following statement:

```
#include <HeaderFile.h>
```

The following memory mapping services are provided:

madvise	Advises the system of a process' expected paging behavior.
mincore	Determines residency of memory pages.
mmap	Maps an object file onto virtual memory.
mprotect	Modifies access protections of memory mapping.
msync	Synchronizes a mapped file with its underlying storage device.
munmap	Unmaps a mapped memory region.

Chapter 20. Packaging Software for Installation

This article provides information about preparing applications to be installed using the AIX **installp** command.

This section describes the format and contents of the software product installation package that must be supplied by the product developer. It gives a description of the required and optional files that are part of a software installation or update package.

An AIX software product installation package is an AIX backup-format file containing the files of the software product, required installation control files, and optional installation customization files. The **installp** command is used to install and update software products.

An *installation package* contains one or more separately installable, logically-grouped units called *filesets*. Each fileset in a package must belong to the same product.

A *fileset update* or *update package* is a package containing modifications to an existing fileset.

Throughout this article, the term *standard system* is used to refer to a system that is not configured as a diskless system.

This article contains the following main sections:

Note: Starting in AIX Version 4.3, a new service is available to application developers. If your online documentation is written in HTML, you should register your documentation with the Documentation Library Service during installation. Your documents will then appear in the Documentation Library GUI so that users can search, navigate, and read your online documents. The service can also be launched from within your application to provide a custom GUI for users to read your application's documents. For information on how to build your install package to use this service, see "Chapter 21. Documentation Library Service" on page 607 before you build your install package.

Installation Procedure Requirements

- Installation must not require user interaction. Product configuration requiring user interaction must occur before or after installation.

- All installations of or updates to interdependent filesets must be able to be performed during a single installation.
- No system restart should be required for installation. The installation may stop portions of the system related to the installation, and a system restart may be required after installation in order for the installation to take full effect.

Package Control Information Requirements

- The control information must specify all installation requirements the filesets have on other filesets.
- The control information must specify all file system size requirements for the fileset installation.

Package Partitioning Requirements

- In order to support client workstations, machine-specific portions of the package (the *root part*) must be separated from the machine-shareable portions of the package (the *usr part*). The usr part of the package contains files that reside in the **/usr** file system.
- Installation of the root part of the package must not modify any files in the **/usr** file system. The **/usr** file system is not writable during installation of the root part of a client system.

Software Product Packaging Parts

In order to support installation in the client/server environment, the installation packaging is divided in the following parts:

usr
Contains the part of the product that can be shared among several machines with compatible hardware architectures. For a standard system, these files are stored in the **/usr** file tree.

root
Contains the part of the product that cannot be shared among machines. Each client must have its own copy. Most of this software requiring a separate copy for each machine is associated with the configuration of the machine or product. For a standard system, files in the root part are stored in the root (/) file tree. The root part of a fileset must be in the same package as the usr part of the fileset. If a fileset contains a root part, it must also contain a usr part.

share
Contains the part of the product that can be shared among several machines, even if the machines have a different hardware architecture. The share part of the product can include non-executable files, such as documentation and data files. For a standard system, files are stored in the **/usr/share** file tree. A share part fileset package must be separately packaged from usr and root parts, and the fileset name cannot be the same as a fileset which has usr or root parts.

Sample File System Guide for Package Partitioning

Following is a brief description of some AIX file systems and directories. You can use this as a guide for splitting a product package into root, usr, and share parts.

Some root-part directories and their contents:

/dev
Local machine device files

/etc	Machine configuration files such as **hosts** and **passwd**
/sbin	System utilities needed to boot the system
/var	System-specific data files and log files

Some usr-part directories and their contents:

/usr/bin	User commands and scripts
/usr/sbin	System administration commands
/usr/include	Include files
/usr/lib	Libraries, non-user commands, and architecture-dependent data

Some share-part directories and their contents:

/usr/share/dict	Dictionary files
/usr/share/man	Manual pages

Format of a Software Package

An installation or update package must be a single file in backup format that can be restored by the **installp** command during installation. This file can be distributed on tape, diskette, or CD-ROM. See "Format of Distribution Media" on page 595 for information about the format used for product packages on each type of media.

Package and Fileset Naming Conventions

Use the following conventions when naming a software package and its filesets:

A package name (*PackageName*) should begin with the product name. All package names must be unique.

A fileset name has the form:

PackageName[[.*SubProduct*] **.** *Option*]

If a package has only one installable fileset, the fileset name may be the same as the *PackageName.*

SubProduct identifies the set of filesets within the package.

Option further describes the fileset and may contain a fileset extension.

A fileset name contains more than one character and begins with a letter or an underline (_). Subsequent characters can be letters, digits, underlines, dots (**.**), plus signs (**+**), minus signs (**-**), exclamations (**!**), tildes (**~**), percent signs (**%**), and carets (**^**). A fileset name cannot end with a dot. All characters in a fileset name are ASCII characters. The maximum length for a fileset name is 144 bytes. All fileset names must be unique within the package.

Fileset Extension Naming Conventions

The following list provides some fileset extension naming conventions:

Extension	Fileset Description
.adt	Application development toolkit

.com	Common code required by similar filesets
.compat	Compatibility code that may be removed in a future release
.data	Share portion of a package
.diag	Diagnostics support
.fnt	Fonts
.info. *Language*	InfoExplorer databases for a particular language
.help. *Language*	Common Desktop Environment (CDE) help files for a particular language
.loc	Locale
.mp	Multiprocessor-specific code
.msg. *Language*	Message files for a particular language
.rte	Run-time environment or minimum set for a product
.ucode	Microcode
.up	Uniprocessor-specific code

Special Naming Considerations for Device Driver Packaging

The **cfgmgr** (configuration manager command) automatically installs software support for detectable devices that are available on the installation media and packaged with the following naming convention:

devices.*BusTypeID***.***CardID*

BusTypeID	Specifies the type of bus to which the card attaches (for example, **mca** for Micro Channel Adapter
CardID	Specifies the unique hexadecimal identifier associated with the card type

For example, a token-ring device attaches to the Micro Channel and is identified by the configuration manager as having a unique card identifier of `8fc8`. The package of filesets associated with this token-ring device is named `devices.mca.8fc8`. A microcode fileset within this package is named `devices.mca.8fc8.ucode`.

Special Naming Considerations for Message Catalog Packaging

A user installing a package can request the message catalogs be installed automatically. When this request is made, the system automatically installs message filesets for the primary language if the message filesets are available on the installation media and packaged with the following naming convention:

*Product***.msg.***Language***[.***SubProduct***]**

The optional *.SubProduct* suffix is used when a product has multiple message catalog filesets for the same language, each message catalog fileset applying to a different *SubProduct*. You can choose to have one message fileset for an entire product.

For example, the `Super.Widget` product has a `plastic` and a `metal` set of fileset options. All `Super.Widget` English U.S. message catalogs can be packaged in a single fileset named `Super.Widget.msg.en_US`. If separate message catalog filesets are needed for the `plastic` and `metal` options, the English U.S. message catalog filesets would be named `Super.Widget.msg.en_US.plastic` and `Super.Widget.msg.en_US.metal`.

Note: A message fileset that conforms to this naming convention MUST contain an installed-requisite on another fileset in the product in order to avoid accidental automatic installation of the message fileset.

File Names

Files delivered with the software package cannot have names containing commas or colons. Commas and colons are used as delimiters in the control files used during the software installation process. File names can contain non-ASCII characters.

Fileset Revision Level Identification

Fileset Level Overview

The fileset level is referred to as the *level* or alternatively as the *v.r.m.f* or *VRMF* and has the form:

Version.Release.ModificationLevel.FixLevel[.FixID]

`Version`	A numeric field of 1 to 2 digits that identifies the version number.
`Release`	A numeric field of 1 to 2 digits that identifies the release number.
`ModificationLevel`	A numeric field of 1 to 4 digits that identifies the modification level.
`FixLevel`	A numeric field of 1 to 4 digits that identifies the fix level.
`FixID`	A character field of 1 to 9 characters identifying the fix identifier. The `FixID` is used by Version 3.2-formatted fileset updates only.

A *base fileset installation level* is the full initial installation level of a fileset. This level contains all files in the fileset, as opposed to a fileset update, which may contain a subset of files from the full fileset.

All filesets in a software package should have the same fileset level, though it is not required for AIX Version 4.1-formatted packages.

For all new levels of a fileset, the fileset level must increase. The **installp** command uses the fileset level to check for a later level of the product on subsequent installations.

Fileset level precedence reads from left to right (for example, `3.2.0.0` is a newer level than `2.3.0.0`).

Fileset Level Rules and Conventions for AIX Version 4.1-Formatted Filesets

The following conventions and rules have been put in place in order to simplify the software maintenance for product developers and customers:

- A base fileset installation level should have a fix level of `0` (zero).
- A base fileset installation level package must contain the functionality provided in other installation packages for that fileset with lower fileset levels. For example, the `Plan.Day` level `2.1` fileset must contain the functionality provided in the `Plan.Day` level `1.1` fileset.
- A fileset update must have either a non-zero modification level or a non-zero fix level.
- A fileset update must have the same version and release numbers as the base fileset installation level to which it is to be applied.
- Unless otherwise specified in the software package, a fileset update with a non-zero fix level must be an update to the fileset with the same version number, release number, and modification level and a zero fix level. Providing information in the requisite section of the **lpp_name** file causes an exception to this rule.

- Unless otherwise specified in the software package, a fileset update with a non-zero modification level and a zero fix level must be an update to the fileset with the same version number and release number and a zero modification level. Providing information in the requisite section of the **lpp_name** file causes an exception to this rule.
- A fileset update must contain the functionality of the fileset's previous updates that apply to the same fileset level.

Compatibility Information For Version 3.2-Formatted Fileset Updates

The fix identifier is required for all 3.2-formatted update packages. It is not allowable in any other types of software packages.

The fix identifier is not allowed as part of the product level for a base fileset installation level.

The fix identifier contains ASCII characters only. The first character must be a letter. Subsequent characters can be letters or digits. All fix identifiers must be unique within a product.

Fix identifiers beginning with `U4` are reserved for the AIX operating system manufacturer.

Contents of a Software Package

This section describes the files contained in an installation or update package. File path names are given for installation package types. For update packages, wherever *PackageName* is part of the path name, it is replaced by *PackageName*/*FilesetName*/*FilesetLevel* (or for the obsolete 3.2->formatted updates, *PackageName*/**inst**_*FixID* where *FixID* is the fix ID of the update).

The usr part of an installation or update package contains the following installation control files:

- **./lpp_name**

This file provides information about the software package to be installed or updated. For performance reasons, the **lpp_name** file should be the first file in the backup-format file that makes up a software installation package. See "The lpp_name Package Information File" on page 573 for more information.

- **./usr/lpp/***PackageName***/liblpp.a**

This archive file contains control files used by the installation process for installing or updating the usr part of the software package. See The liblpp.a Installation Control File ("The liblpp.a Installation Control Library File" on page 584) for information about files contained in this archive library.

- All files, backed up relative to root, that are to be restored for the installation or update of the usr part of the software product.

If the installation or update package contains a root part, the root part contains the following files:

- **./usr/lpp/***PackageName***/inst_root/liblpp.a**

This library file contains control files used by the installation process for installing or updating the root part of the software package.

- All files that are to be restored for the installation or update of the root part of the software package. For a base fileset installation level. these files must be backed up relative to **./usr/lpp/***PackageName***/inst_root**.

If the software product has a share part, it must be packaged in a separate installation package from the usr and root parts. The backup format file that makes up an installation or update package for the share part of a software product must contain the following files:

- **./lpp_name**

This file provides information about the share part of the software package to be installed or updated.

- **./usr/share/lpp/***PackageName***/liblpp.a**

This library file contains control files used by the installation process for installing or updating the share part of the software package.

- All files, backed up relative to root, that are to be restored for the installation or update of the share part of the software package.

Example Contents of a Software Package

The `farm.apps` package contains the `farm.apps.hog 4.1.0.0` fileset. The `farm.apps.hog 4.1.0.0` fileset delivers the following files:

```
/usr/bin/raisehog (in the usr part of the package)
/usr/sbin/sellhog
 (in the usr part of the package)

/etc/hog
 (in the root part of the package)
```

The `farm.apps` package contains at least the following files:

```
./lpp_name
./usr/lpp/farm.apps/liblpp.a
./usr/lpp/farm.apps/inst_root/liblpp.a
./usr/bin/raisehog
./usr/sbin/sellhog
./usr/lpp/farm.apps/inst_root/etc/hog
```

Fileset update `farm.apps.hog 4.1.0.3` delivers updates to the following files:

```
/usr/sbin/sellhog
/etc/hog
```

The fileset update package contains the following files:

```
./lpp_name
./usr/lpp/farm.apps/farm.apps.hog/4.1.0.3/liblpp.a
./usr/lpp/farm.apps/farm.apps.hog/4.1.0.3/inst_root/liblpp.a
./usr/sbin/sellhog
./usr/lpp/farm.apps/farm.apps.hog/4.1.0.3/inst_root/etc/hog
```

Note that the file from the root part of the package was restored under an **inst_root** directory. Files installed for the machine-dependent root part of a package are restored relative to an **inst_root** directory. This facilitates installation of machine-specific files in the root file systems of multiple systems. The root part files are installed into the root portions of systems by copying the files from the **inst_root** directory. This allows multiple machines to share a common machine-independent usr part.

The lpp_name Package Information File

Each software package must contain the **lpp_name** package information file. The **lpp_name** file gives the **installp** command information about the package and each fileset in the package. Refer to the figure for an example **lpp_name** file for a fileset update package. The numbers and arrows in the figure refer to fields that are described in the table that follows.

The following table defines the fields in the **lpp_name** file.

Fields in the lpp_name File			
Field Name	**Format**	**Separator**	**Description**

1. Format	Integer	White space	Indicates the release level of **installp** for which this package was built. The values are: • 1 - AIX Version 3.1 • 3 - Version 3.2 • 4 - AIX Version 4.1
2. Platform	Character	White space	Indicates the platform for which this package was built. The only available value is R.
3. Package Type	Character	White space	Indicates whether this is an installation or update package and what type. The values are: • I - Installation • S - Single update • SR - Single update required • ML - Maintenance level update The following types are valid for Version 3.2-formatted update packages only: • G -Single update required • M -Maintenance packaging update • MC -Cumulative packaging update • ME -Enhancement packaging update
4. Package Name	Character	White space	The name of the software package (*PackageName*).
	{	New line	Indicates the beginning of the repeatable sections of fileset-specific data.
5. Fileset name	Character	White space	The complete name of the fileset. This field begins the heading information for the fileset or fileset update.
6. Level	Shown in Description column	White space	The level of the fileset to be installed. The format is: *Version.Release.ModificationLevel.FixLevel .FixID* should be appended for Version 3.2-formatted updates only.
7. Diskette Volume	Integer	White space	Indicates the diskette volume number where the fileset is located, if shipped on diskette.

8. Bosboot	Character	White space	Indicates whether a bosboot is needed following the installation. The values are: • N - Do not invoke bosboot • b - Invoke bosboot
9. Content	Character	White space	Indicates the parts included in the fileset or fileset update. The values are: • B -usr and root part • H -share part • U -usr part only
10. Language	Character	White space	Not used.
11. Description	Character	# or new line	Fileset description.
12. Comments	Character	New line	(Optional) Additional comments.
	[	New line	Indicates the beginning of the body of the fileset information.
13. Requisite information	Described following table	New line	(Optional) Installation dependencies the fileset has on other filesets and fileset updates. See the section following this table for detailed description.
	%	New line	Indicates the separation between requisite and size information.
14. Size and License Agreement information	Described later in this chapter	New line	Size requirements by directory and license agreement information. See Size and License Agreement Information Section later in this article for detailed description.
	%	New line	Indicates the separation between size and supersede information.
15. Supersede information	Described later in this chapter	New line	(Optional) Information on what the fileset replaces. This field should exist in Version 3.2-formatted updates only. See Supersede Information Section later in this article for detailed description.
	%	New line	Indicates the separation between supersede and licensing information.

16. Fix information	Described later in article	New line	Information regarding the fixes contained in the fileset update. See Fix Information Section later in this article for detailed description.
	]	New line	Indicates the end of the body of the fileset information.
	}	New line	Indicates the end of the repeatable sections of fileset-specific information.

```
      1 2 3    4
      | | |    |          6      7 89 10     11         12        4 R  S  farm.apps { |      | || |     |          |
5 --> farm.apps.hog  04.01.0000.0003 1 N U en_US Hog Utilities # ...
[
13--> *ifreq bos.farming.rte (4.2.0.0) 4.2.0.15
%
14--> /usr/sbin 48
14--> /usr/lpp/farm.apps/farm.apps.hog/4.1.0.3 280
14--> /usr/lpp/farm.apps/farm.apps.hog/inst_root/4.1.0.3.96
14--> /usr/lpp/SAVESPACE 48
14--> /lpp/SAVESPACE 32
14--> /usr/lpp/bos.hos/farm.apps.hog/inst_root/4.1.0.3/ etc 32
15--> INSTWORK 348 128
%
%
16--> iFOR/LS_vendor_id
17--> iFOR/LS_product_id
18--> iFOR/LS_product_version
%
19--> IX51366 Hogs producing eggs.
19--> IX81360 Piglets have too many ears.
]
}
```

Example lpp_name File

Requisite Information Section

The requisite information section contains information about installation dependencies on other filesets or fileset updates. Each requisite listed in the requisite section must be satisfied according to the requisite rules in order to apply the fileset or fileset update.

Before any installing or updating occurs, the **installp** command compares the current state of the filesets to be installed with the requirements listed in the **lpp_name** file. If the **-g** flag was specified with the **installp** command, any missing requisites are added to the list of filesets to be installed. The filesets are ordered for installation according to any prerequisites. Immediately before a fileset is installed, the **installp** command again checks requisites for that fileset. This check verifies that all requisites installed earlier in the installation process were installed successfully and that all requisites are satisfied.

In the following descriptions of the different types of requisites, *RequiredFilesetLevel* represents the minimum fileset level that satisfies the requirements. Except when explicitly blocked by mechanisms described in “Supersede Information Section” on page 582, newer levels of a fileset satisfy requisites on an earlier level. For example, a requisite on the `plum.tree 2.2.0.0` fileset is satisfied by the `plum.tree 3.1.0.0` fileset.

Prerequisite

Syntax ***prereq** *Fileset RequiredFilesetLevel*

Alternate Syntax *Fileset RequiredFilesetLevel*

Description A prerequisite indicates that the specified fileset must be installed at the specified fileset level or at a higher level for the current fileset to install successfully. If a prerequisite fileset is scheduled to be installed, the **installp** command orders the list of filesets to install to make sure the prerequisite is met.

A fileset update contains an implicit prerequisite to its base-level fileset. If this implicit prerequisite is not adequate, you must specify explicitly a different prerequisite. The *Version* and *Release* of the update and the implicit prerequisite are the same. If the *FixLevel* of the update is `0`, the *ModificationLevel* and the *FixLevel* of the implicit prerequisite are both `0`. Otherwise, the implicit prerequisite has a *ModificationLevel* that is the same as the *ModificationLevel* of the update and a *FixLevel* of `0`. For example, a 4.1.3.2 level update requires its 4.1.3.0 level to be installed before the update installation. A 4.1.3.0 level update requires its 4.1.0.0 level to be installed before the update installation.

Co-requisite

Syntax ***coreq** *Fileset RequiredFilesetLevel*

Description A co-requisite indicates that the specified fileset must be installed for the current fileset to function successfully. At the end of the installation process, the **installp** command issues warning messages for any unmet co-requisites. A co-requisite is most commonly used for a fileset within the same package. A prerequisite on a fileset within the same package is not guaranteed.

If-requisite

Syntax ***ifreq** *Fileset* [(*InstalledFilesetLevel*)] *RequiredFilesetLevel*

Description An if-requisite indicates that the specified fileset is required to be at *RequiredFilesetLevel* only if the fileset is installed at *InstalledFilesetLevel*. This is most commonly used to coordinate dependencies between fileset updates. The following example shows an if-requisite:

```
*ifreq A.obj (1.1.0.0) 1.1.2.3
```

If the `A.obj` fileset is not already installed, this example does not cause it to be installed. If the `A.obj` fileset is already installed at any of the following levels, this example does not cause the `1.1.2.3` level to be installed:

`1.1.2.3`	This level matches the *RequiredFilesetLevel*.
`1.2.0.0`	This level is a different base fileset level.
`1.1.3.0`	This level supersedes the *RequiredFilesetLevel*.

If the `A.obj` fileset is already installed at any of the following levels, this example causes the `1.1.2.3` level to be installed:

`1.1.0.0`	This level matches the *InstalledFilesetLevel*.
`1.1.2.0`	This level is the same base level as the *InstalledFilesetLevel* and a lower level than the *RequiredFilesetLevel*.

The (*InstalledFilesetLevel*) parameter is optional. If it is omitted, the *Version* and *Release* of the *InstalledFilesetLevel* and the *RequiredFilesetLevel* are assumed to be the same. If the *FixLevel* of the *RequiredFilesetLevel* is `0`, the *ModificationLevel* and the *FixLevel* of the *InstalledFilesetLevel* are both `0`. Otherwise, the *InstalledFilesetLevel* has a *ModificationLevel* that is the same as the *ModificationLevel* of the *RequiredFilesetLevel* and a *FixLevel* of `0`. For example, if the *RequiredFilesetLevel* is `4.1.1.1` and no *InstalledFilesetLevel* parameter is supplied, the *InstalledFilesetLevel* is `4.1.1.0`. If the *RequiredFilesetLevel* is `4.1.1.0` and no *InstalledFilesetLevel* parameter is supplied, the *InstalledFilesetLevel* is `4.1.0.0`.

Installed-requisite

Syntax *instreq *Fileset RequiredFilesetLevel*

Description An installed-requisite indicates that the specified fileset should be installed automatically only if its corresponding fileset is already installed or is on the list of filesets to install. An installed-requisite also is installed if the user explicitly requests that it be installed. A fileset update can not have an installed-requisite. Because a fileset containing the message files for a particular package should not be installed automatically without some other part of the package being installed, a message fileset should contain an installed-requisite for another fileset in its package.

Group Requisite

Syntax >*Number* { *RequisiteExpressionList* }

Description A group requisite indicates that different requisite conditions can satisfy the requisite. A group requisite can contain prerequisites, co-requisites, if-requisites, and nested group requisites. The *Number* preceding the { *RequisiteExpressionList* } identifies how many of the items in the *RequisiteExpressionList* are required. For example, `>2` states that at least three items in the *RequisiteExpressionList* are required.

Alternate Requisite Syntax for AIX Versions 3.1 and 3.2 Compatibility

For compatibility with AIX Versions 3.1- and 3.2-formatted packages, you can specify *RequiredFilesetLevel* using an alternate syntax. This alternate syntax cannot be used for an AIX Version 4.1-formatted fileset update that contains a prerequisite on another level of the same fileset.

The alternate syntax consists of logical expressions using the letters **v** (version number), **r** (release number), **m** (modification level), **f** (fix level), and **p** (Version 3.2-format update fix ID) and the symbols <, =, and >. If multiple conditions apply to a field, an `o` (*or* symbol) is used to identify an alternate acceptable condition for the previously mentioned field. The following example specifies a prerequisite on the `old.syntax` fileset with the version number 3 and the release level greater than or equal to 2.

```
*prereq old.syntax v=3 r=2 o>2
```

Because the **installp** command interprets the specified requisite as the minimum fileset level that will satisfy the requisite, `o>2` can not be specified the following example. The following example is equivalent to the preceding example:

```
*prereq old.syntax 3.2.0.0
```

Alternate syntax containing a fix ID is handled differently than in Version 3.2. If the *RequiredFilesetLevel* expression contains more information than just a fix ID, either a newer base level of the fileset or the fix (defined by the fix ID) satisfies the requisite. (A supersede entry can still block a newer base level from satisfying the requisite. See “Supersede Information Section” on page 582 for more information about supersede entries.) For example, the `old.syntax 1.3.0.0` fileset satisfies the following requisite:

```
*prereq old.syntax v=1 r=2 p=U412345
```

The alternate syntax also can be used to define a requisite that is lower than a certain level. For example, the `old.syntax` fileset at a lower level than `1.3.0.0` satisfies the following requisite:

```
*prereq old.syntax v=1 r<3
```

Requisite Information Section Examples

1. The following example illustrates the use of co-requisites. The `book.create 12.30.0.0` fileset cannot function without the `layout.text 1.1.0.0` and `index.generate 2.3.0.0` filesets installed, so the requisite section for `book.create 12.30.0.0` contains:

   ```
   *coreq layout.text 1.1.0.0
   *coreq index.generate 2.3.0.0
   ```

   The `index.generate 3.1.0.0` fileset satisfies the `index.generate` requisite, because `3.1.0.0` is a newer level than the required `2.3.0.0` level.

2. The following example illustrates the use of the more common requisite types. Fileset `new.fileset.rte 1.1.0.0` contains the following requisites:

   ```
   *prereq database 1.2.0.0
   *coreq spreadsheet 1.3.1.0
   *ifreq wordprocessorA (4.1.0.0) 4.1.1.1
   *ifreq wordprocessorB 4.1.1.1
   ```

   The `database` fileset must be installed at level `1.2.0.0` or higher before the `new.fileset.rte` fileset can be installed. If `database` and `new.fileset.rte` are installed in the same installation session, the installation program will install the `database` fileset before the `new.fileset.rte` fileset.

   The `spreadsheet` fileset must be installed at level `1.3.1.0` or higher for the `new.fileset.rte` fileset to function properly. The `spreadsheet` fileset does not need to be installed before the `new.fileset.rte` fileset is installed, provided both are installed in the same installation session. If an adequate level of the `spreadsheet` fileset is not installed by the end of the installation session, a warning message will be issued stating that the co-requisite is not met.

   If the `wordprocessorA` fileset is installed (or being installed with `new.fileset.rte`) at level `4.1.0.0`, the `wordprocessorA` fileset update must be installed at level `4.1.1.1` or higher.

   If the `wordprocessorB` fileset is installed (or being installed with `new.fileset.rte`) at level `4.1.1.0`, the `wordprocessorB` fileset update must be installed at level `4.1.1.1` or higher.

3. The following example illustrates an installed-requisite. Fileset `Super.msg.fr_FR.Widget` at level `2.1.0.0` contains the following install-requisite:

   ```
   Super.Widget 2.1.0.0
   ```

   The `Super.msg.fr_FR.Widget` fileset can not be installed automatically when the `Super.Widget` fileset is not installed. The `Super.msg.fr_FR.Widget` fileset can be installed explicitly when the `Super.Widget` fileset is not installed.

4. The following example illustrates a group requisite. At least one of the prerequisite filesets listed must be installed (both can be installed). If installed, the `spreadsheet_1` fileset must be at least at level `1.2.0.0` and the `spreadsheet_2` fileset must be at least at level `1.3.0.0`.

   ```
   >0 {
   *prereq spreadsheet_1 1.2.0.0
   *prereq spreadsheet_2 1.3.0.0
   }
   ```

5. The following example illustrates use of the alternate requisite syntax for compatibility with AIX Versions 3.1- and 3.2-formatted packages. The following requisite expressions are all equivalent to the expression `*prereq database 1.2.0.0` and will be satisfied as described in Example 2.

```
*prereq database v=1 r=2
*prereq database v=1 r=2 o=3
*prereq database v=1 r=2 o>2
*prereq database v=1 r=2 m=0 f=0
```

Note: The preceding examples stated that levels higher than the level specified in the requisite expression satisfy the requisite. This is not true when the required fileset indicates it has broken compatibility by including a barrier entry in the supersede information section of the **lpp_name** file. See "Supersede Information Section" on page 582 for more information.

Size and License Agreement Information Section

The size and license agreement information section contains information about the disk space and license agreement requirements for the fileset.

Size Information

This information is used by the installation process to ensure that enough disk space is available for the installation or update to succeed. Size information has the form:

Directory PermanentSpace **[***TemporarySpace***]**

Additionally, the product developer can specify **PAGESPACE** or **INSTWORK** in the full-path name field to indicate disk space requirements for paging space and work space needed in the package directory during the installation process.

Directory
: The full path name of the directory that has size requirements.

PermanentSpace
: The size (in 512-byte blocks) of the permanent space needed for the installation or update. Permanent space is space that is needed after the installation completes. This field has a different meaning in the following cases:

If *Directory* is **PAGESPACE**, *PermanentSpace* represents the size of page space needed (in 512-byte blocks) to perform the installation.

If *Directory* is **INSTWORK**, *PermanentSpace* represents the number of 512-byte blocks needed for extracting control files used during the installation. These control files are the files that are archived to the **liblpp.a** file.

TemporarySpace
: The size (in 512-byte blocks) of the temporary space needed for the installation only. Temporary space is released after the installation completes. The *TemporarySpace* value is optional. An example of temporary space is the space needed to relink an executable object file. Another example is the space needed to archive an object file into a library. To archive into a library, the **installp** command makes a copy of the library, archives the object file into the copied library, and moves the copied library over the original library. The space for the copy of the library is considered temporary space.

When *Directory* is **INSTWORK**, *TemporarySpace* represents the number of 512-byte blocks needed for the unextracted **liblpp.a** file.

The following example shows a size information section:

```
/usr/bin        30
/lib            40 20
PAGESPACE       10
INSTWORK        10  6
```

Because it is difficult to predict how disk file systems are mounted in the file tree, the directory path name entries in the size information section should be as specific as possible. For example, it is better to have an entry for **/usr/bin** and one for **/usr/lib** than to have a combined entry for **/usr**, because **/usr/bin** and **/usr/lib** can exist on different file systems that are both mounted under **/usr**. In general, it is best to include entries for each directory into which files are installed.

For an update package only, the size information must include any old files (to be replaced) that will move into the **save** directories. These old files will be restored if the update is later rejected. In order to indicate these size requirements, an update package must specify the following special directories:

/usr/lpp/SAVESPACE	The **save** directory for usr part files. By default, the usr part files are saved in the **/usr/lpp/***PackageName*/*FilesetName*/*FilesetLevel***.save** directory.
/lpp/SAVESPACE	The **save** directory for root part files. By default, the root part files are saved in the **/lpp/***PackageName*/*FilesetName*/*FilesetLevel***.save** directory.
/usr/share/lpp/SAVESPACE	The **save** directory for share part files. By default, the share part files are saved in the **/usr/share/lpp/***PackageName***inst**_*FixID***.save** directory.

The following save directories are used for obsolete 3.2-formatted fileset updates only:

/usr/lpp/*PackageName***inst**_*FixID***.save**	Directory for usr part files.
/lpp/*PackageName***inst**_*FixID***.save**	Directory for root part files.
/usr/share/lpp/*PackageName***inst**_*FixID***.save**	Directory for share part files.

License Agreement Information

Products which require users to agree to software license terms before the product can be installed must provide special entries in the size and license agreement information section. There are two forms of license agreement information: license agreement requirement entries indicating that a fileset is governed by a license agreement and license agreement file entries indicating that a package contains a license agreement file.

License agreement files are indicated by a size section entry which begins with **LAF**<**%**locale_spec>**/**, where **LAF** stands for **L**icense **A**greement **F**ile, and locale_spec specifies the locale for which the file is encoded. If the locale spec is not specified, then the agreement file will be assumed to be nontranslated and encoded as ASCII. If the license agreement file is translated, the locale name must be part of the path so that the requirement entry may be associated with the file.
The remainder of the string is a path designation which uniquely identifies a particular license file which is included in the package. Each time that the license text changes, the fullpath of the license must also change. Licenses are not modifiable once they have been delivered, since the terms associated with preexisting products cannot be modified. Only one fileset in the package is required to carry the license agreement file information, though it may appear in the size and license information section of more than one. It is desirable to only carry the license agreement file information for one fileset since the information would otherwise be redundant. Note: the license agreement file should be in neither the apply list nor the inventory for a fileset in order to prevent the removal of the license agreement when the fileset is removed. If there are unique directories required for the license files, those must be included in the apply list and inventory in order to properly control the permissions associated with the directories.

The second field in the license agreememt file indicator is the size in 512-byte blocks of the license agreement file, rounded up to multiples of 8 to reflect a 4096-byte block size for a standard JFS filesystem.

License agreement requirements are indicated by a size and license information section entry which begins with **LAR/**, where **LAR** stands for **L**icense **A**greement **R**equirement. The remainder of the string is the filepath which uniquely identifies the particular license. The license agreement requirement entry will have an additional second field of 0 to indicate that no additional size requirements exist. The actual size information will be derived from the license agreement file size indication.

Internationalization: Each available translated license agreement file must be listed in the size and license information section of one of the filesets in the package. Translated license agreement files must include a locale designation as part of the full pathname.
License agreement requirement entries indicate the base pattern of the license files associated with license agreement file entries. A **%L** pattern will designate the locale substitution string which will be applied to identify which particular license agreement file to use.

In the following example, product `IcedTea` contains a license information file which has been translated into English, Japanese, and German. The fileset `IcedTea.rte` both requires and provides a license agreement. The basic form of the license file provided for `IcedTea` is `/usr/opt/IcedTea/license/`*`LANG`*`/license.txt`. The size and license information section for `IcedTea.rte` would include the following entries:

```
LAF%en_US/usr/opt/IcedTea/license/en_US/license.txt 8
LAF%ja_JP/usr/opt/IcedTea/license/ja_JP/license.txt 8
LAF%de_DE/usr/opt/IcedTea/license/de_DE/license.txt 8
LAR/usr/opt/IcedTea/license/%L/license.txt 0
```

The following files would be included in the package containing `IcedTea.rte`:

```
./usr/opt/IcedTea/license/en_US/license.txt
./usr/opt/IcedTea/license/ja_JP/license.txt
./usr/opt/IcedTea/license/de_DE/license.txt
```

The license agreement files may be carried in separate packages from the packages containing the license agreement requirements. This enables shipping translations separately and also allows for a product consisting of multiple packages to only ship the license agreement files in a single package. It is preferable to limit the number of filesets in a product as much as possible. If there is a common fileset which is a prerequisite for the other filesets in a product, the license agreement requirement should just be associated with that one fileset.

Supersede Information Section

The supersede information section indicates the levels of a fileset or fileset update for which this fileset or fileset update may (or may not) be used as a replacement. Supersede information is optional and is only applicable to AIX Version 4.1-formatted fileset base installation packages and Version 3.2-formatted fileset update packages.

A newer fileset supersedes any older version of that fileset unless the supersedes section of the **lpp_name** file identifies the latest level of that fileset it supersedes. In the rare cases where a fileset does not supersede all earlier levels of that fileset, the **installp** command does not use the fileset to satisfy requisites on levels older than the level of the fileset listed in the supersedes section.

A fileset update supersedes an older update for that fileset only if it contains all of the files, configuration processing, and requisite information contained in the older fileset update. The **installp** command determines that a fileset update supersedes another update for that fileset in the following conditions:

- The version, release, and modification levels for the updates are equal, the fix levels are both non-zero, and the update with the higher fix level does not contain a prerequisite on a level of the fileset greater than or equal to the level of the update with the lower fix level.

- The version and release levels for the updates are equal, and the update with the higher modification level does not contain a prerequisite on a level of the fileset greater than or equal to the level of the update with the lower modification level.

For example, the fileset update `farm.apps.hog 4.1.0.1` delivers an update of `/usr/sbin/sellhog`. Fileset update `farm.apps.hog 4.1.0.3` delivers updates to the `/usr/sbin/sellhog` file and the `/etc/hog` file. Fileset update `farm.apps.hog 4.1.1.2` delivers an update to the `/usr/bin/raisehog` file.

Update `farm.apps.hog 4.1.0.3` supersedes `farm.apps.hog 4.1.0.1` because it delivers the same files and applies to the same level, `farm.apps.hog 4.1.0.0`.

Update `farm.apps.hog 4.1.1.2` does not supersede either `farm.apps.hog 4.1.0.3` or `farm.apps.hog 4.1.0.1` because it does not contain the same files and applies to a different level, `farm.apps.hog 4.1.1.0`. Update `farm.apps.hog 4.1.1.0` supersedes `farm.apps.hog 4.1.0.1` and `farm.apps.hog 4.1.0.3`.

Supersede Section for Fileset Installation Levels (Base Levels)

An AIX Version 4.1-formatted fileset installation package can contain the following supersede entries:

Barrier Entry	Identifies the fileset level where a major incompatibility was introduced. Such an incompatibility keeps the current fileset from satisfying requisites to levels of the fileset earlier than the specified level.
Compatibility Entry	Indicates the fileset can be used to satisfy the requisites of another fileset. A compatibility entry is used when a fileset has been renamed or made obsolete. Only one fileset can supersede a given fileset. You may specify only one compatibility entry for each fileset.

The **lpp_name** file can contain at most one barrier and one compatibility entry for a fileset.

A barrier entry consists of the fileset name and the fileset level when the incompatibility was introduced. A barrier entry is necessary for a fileset only in the rare case that a level of the fileset has introduced an incompatibility such that functionality required by dependent filesets has been modified or removed to such an extent that requisites on previous levels of the fileset are not met. A barrier entry must exist in all subsequent versions of the fileset indicating the latest level of the fileset that satisfies requisites by dependent filesets.

For example, if a major incompatibility was introduced in fileset `Bad.Idea 6.5.6.0`, the supersede information section for each `Bad.Idea` fileset installation package from fileset level `6.5.6.0` onward would contain a `Bad.Idea 6.5.6.0` barrier entry. This barrier entry would prevent a requisite of `Bad.Idea 6.5.4.0` from being satisfied by any levels of `Bad.Idea` greater than or equal to `6.5.6.0`.

A compatibility entry consists of a fileset name (different from the fileset in the package) and a fileset level. The fileset level identifies the level at which requisites to the specified fileset (and earlier levels of that fileset) are met by the fileset in the installation package. The compatibility is useful when the specified fileset is obsolete or has been renamed, and the function from the specified fileset is contained in the current fileset. The fileset level in the compatibility entry should be higher than any level expected to exist for the specified fileset.

For example, the `Year.Full 19.91.0.0` fileset is no longer delivered as a unit and is instead broken into several smaller, individual filesets. Only one of the smaller resulting filesets, perhaps `Winter 19.94.0.0`, should contain a compatibility entry of `Year.Full 19.94.0.0`. This compatibility entry allows the `Winter 19.94.0.0` fileset to satisfy the requisites of filesets dependent on `Year.Full` at levels `19.94.0.0` and earlier.

Supersede Section for Version 3.2-Formatted Updates

A Version 3.2-formatted fileset update can supersede another update for that fileset if each file and configuration action contained in the older update are contained in the newer update and if each of those filesets can be applied to the same fileset installation level. The **installp** command does not use the fileset level and prerequisite information to determine if a Version 3.2-formatted fileset update supersedes another. Instead, the Version 3.2-formatted fileset update must explicitly list each fix identifier that the update supersedes. The supersedes section for a Version 3.2-formatted fileset update consists of a newline-separated list of entries. Each entry contains the fileset name and the fix identifier of the superseded fileset.

Supersedes Processing

The **installp** command provides the following special features for installing filesets and fileset updates which supersede other filesets or fileset updates:

- If the installation media does not contain a fileset or fileset update that the user requested to install, a superseding fileset or fileset update on the installation media can be installed.

For example, the user invokes the **installp** command with the **-g** flag (automatically install requisites) to install the `farm.apps.hog 4.1.0.2` fileset. If the installation media contains the `farm.apps.hog 4.1.0.4` fileset only, the **installp** command will install the `farm.apps.hog 4.1.0.4` fileset because it supersedes the requested level.

- If the system and the installation media do not contain a requisite fileset or fileset update, the requisite can be satisfied by a superseding fileset or fileset update.
- If an update is requested for installation and the **-g** flag is specified, the request is satisfied by the newest superseding update on the installation media.
- If an update and a superseding update (both on the installation media) are requested for installation, the **installp** command installs the newer update only.

When the **-g** flag is specified with the **installp** command, any update requested for installation (either explicitly or implicitly) is satisfied by the newest superseding update on the installation media. If the user wants to install a particular level of an update, not necessarily the latest level, the user can invoke the **installp** command without the **-g** flag.

In the last case, if a user wishes to apply a certain update and its superseding update from the installation media, the user must do separate **installp** operations for each update level. Note that this kind of operation is meaningless if the two updates are applied and committed (**-ac**). Committing the second update removes the first update from the system.

Fix Information Section

The fix information section is optional and is only applicable to update packages. The fix information section entries contain a fix keyword and a 60-character or less description of the problem fixed. A fix keyword is a 16-character or less identifier corresponding to the fix. Fix keywords beginning with **ix** and **IX** are reserved for use by the AIX operating system manufacturer.

A maintenance level is a fix that is a major update level. AIX periodic preventive maintenance packages are maintenance levels. A maintenance level identifier begins with the name of the software product (not the package), followed by a single dot (**.**) and an identifying level, such as `farm.4.1.1.0`.

The liblpp.a Installation Control Library File

The **liblpp.a** file is an AIX archive file that contains the files required to control the package installation. You can create a **liblpp.a** file for your package using the **ar** command. This section describes many of the files you can put in a **liblpp.a** archive.

Throughout this section, *Fileset* appears in the names of the control files. *Fileset* represents the name of the separate fileset to be installed within the software package. For example, the apply list file is described as *Fileset*.**al**. The apply list file for the `bos.net.tcp.client` option of the `bos.net` software product is `bos.net.tcp.client.al`.

For any files you include in the **liblpp.a** archive file other than the files listed in this section, you should use the following naming conventions:

- If the file is used in the installation of a specific fileset, the file name should begin with the *Fileset*. prefix.
- If the file is used as a common file for several filesets in the same package, the file name should begin with the **lpp.** prefix.

Many files described in this section are optional. An optional file is necessary only if the function the file provides is required for the fileset or fileset update. Unless stated, a file pertains to both full installation packages and fileset update packages.

Data Files Contained in the liblpp.a File

Fileset.**al**	Apply list. This file lists all files to be restored for this fileset. Files are listed one per line with a path relative to root, as in `./usr/bin/pickle`. An apply list file is required if any files are delivered with the fileset or fileset update.
Fileset.**cfginfo**	Special instructions file. This file lists one keyword per line, each keyword indicating special characteristics of the fileset or fileset update. The only currently recognized keyword is **BOOT**, which causes a message to be generated after installation is complete indicating that the system needs to be restarted.
Fileset.**cfgfiles**	List of user-configurable files and processing instructions for use when applying a newer or equal installation level of a fileset that is already installed. Before restoring the files listed in the *Fileset*.**al** file, the system saves the files listed in *Fileset*.**cfgfiles** file. Later, these saved files are processed according to the handling methods specified in the *Fileset*.**cfgfiles** file.
Fileset.**copyright**	Required copyright information file for the fileset. This file consists of the full name of the software product followed by the copyright notices.
Fileset.**err**	Error template file used as input to the **errupdate** command to add or delete entries in the Error Record Template Repository. This file is commonly used by device support software. The **errupdate** command creates a *Fileset*.**undo.err** file for cleanup purposes. See the **errupdate** command for information about the format of the *Fileset*.**err** file.
Fileset.**fixdata**	Optional stanza format file. This file contains information about the fixes contained in a fileset or fileset update.
Fileset.**inventory**	The inventory file. This file contains required software vital product data for the files in the fileset or fileset update. The inventory file is a stanza-format file containing an entry for each file to be installed or updated.
Fileset.**namelist**	List of obsolete filesets that once contained files now existing in the fileset to be installed. This file is used for installation of repackaged software products only.
Fileset.**odmadd** *Fileset*.*.**odmadd**	Stanzas to be added to ODM (Object Data Manager) databases.

Fileset.**rm_inv**
: Remove inventory file. This file is for installation of repackaged software products only and must exist if the fileset is not a direct replacement for an obsolete fileset. This stanza-format file contains names of files that need to be removed from obsolete filesets.

Fileset.**trc**
: Trace report template file. The **trcupdate** command uses this file to add, replace, or delete trace report entries in the **/etc/trcfmt** file. The **trcupdate** command creates a *Fileset*.**undo.trc** file for cleanup purposes. Only the root part of a package can contain *Fileset*.**trc** files.

lpp.acf
: Archive control file for the entire package. This file is needed only when adding or replacing an archive member file to an archive file that already exists on the system. The archive control file consists of lines containing pairs of the member file in the temporary directory as listed in the *Fileset*.**al** file and the archive file that the member belongs to, both listed relative to root as in:

```
./usr/ccs/lib/libc/member.o ./usr/ccs/lib/libc.a
```

lpp.README
: Readme file. This file contains information the user should read before using the software. This file is optional and can also be named **README**, **lpp.doc**, **lpp.instr**, or **lpp.lps.**

productid
: Product identification file. This optional file consists of a single line indicating the product name, the product identifier (20-character limit), and the optional feature number (10-character limit).

Optional Executable Files Contained in the liblpp.a File

The product-specific executable files described in this section are called during the installation process. Unless otherwise noted, file names that end in **_i** are used during installation processing only, and file names that end in **_u** are used in fileset update processing only. All files described in this section are optional and can be either shell scripts or executable object modules. Each program should have a return value of 0 (zero), unless the program is intended to cause the installation or update to fail.

Fileset.**config**
Fileset.**config_u**
: Modifies configuration near the end of the default installation or update process. *Fileset*.**config** is used during installation processing only.

Fileset.**odmdel**
Fileset.*.**odmdel**
: Updates ODM database information for the fileset prior to adding new ODM entries for the fileset. The **odmdel** file naming conventions enables a fileset to have multiple **odmdel** files.

Fileset.**pre_d**
: Indicates whether a fileset may be removed. The program must return a value of 0 (zero) if the fileset may be removed. Filesets are removable by default. The program should generate error messages indicating why the fileset is not removable.

Fileset.**pre_i**
Fileset.**pre_u**
: Runs prior to restoring or saving the files from the apply list in the package, but after removing the files from a previously installed version of the fileset.

Fileset.**pre_rm**
: Runs during a fileset installation prior to removing the files from a previously installed version of the fileset.

Fileset.**post_i**
Fileset.**post_u**
: Runs after restoring the files from the apply list of the fileset installation or update.

Fileset.**unconfig**
Fileset.**unconfig_u**
: Undoes configuration processing performed in the installation or update. *Fileset*.**unconfig** is used during installation processing only.

Fileset.**unodmadd**
: Deletes entries that were added to ODM databases during the installation or update.

Fileset.**unpost_i_0**
Fileset.**unpost_u**
: Undoes processing performed following restoring the files from the apply list in the installation or update.

Fileset.**unpre_i**
Fileset.**unpre_u**
: Undoes processing performed prior to restoring the files from the apply list in the installation or update.

If any of these executable files runs a command that may change the device configuration on a machine, that executable file should check the INUCLIENTS environment variable before running the command. If the INUCLIENTS environment variable is set, the command should not be run. The Network Installation Management (NIM) environment uses the **installp** command for many purposes, some of which require the **installp** command to bypass some of its normal processing. NIM sets the INUCLIENTS environment variable when this normal processing must be bypassed.

If the default installation processing is insufficient for your package, you can provide the following executable files in the **liblpp.a** file. If these files are provided in your package, the **installp** command uses your package-provided files in place of the system default files. Your package-provided files must contain the same functionality as the default files or unexpected results can occur. You can use the default files as models for creating your own files. Use of the default files in place of package-provided files is strongly recommended.

instal
: Used in place of the default installation script **/usr/lib/instl/instal**. The **installp** command calls this executable file if a fileset in an installation package is applied.

lpp.cleanup
: Used in place of the default installation cleanup script **/usr/lib/instl/cleanup**. The **installp** command calls this executable file if a fileset in an installation or update package has been partially applied and must be cleaned up to put the fileset back into a consistent state.

lpp.deinstal
: Used in place of the default fileset removal script **/usr/lib/instl/deinstal**. This executable file must be placed in the **/usr/lpp/***PackageName* directory. The **installp** command calls this executable file if a fileset in an installation package is removed.

lpp.reject
: Used in place of the default installation rejection script **/usr/lib/instl/reject**. The **installp** command calls this executable if a fileset update in an update package is rejected. (The default **/usr/lib/instl/reject** script is a link to the **/usr/lib/instl/cleanup** script.)

update
: Used in place of the default fileset update script **/usr/lib/instl/update**. The **installp** command calls this executable file if a fileset in an update package is applied. (The default **/usr/lib/instl/update** script is a link to the **/usr/lib/instl/instal** script.)

To ensure compatibility with the **installp** command, the **instal** or **update** executable provided with a software package must:

- Process all of the filesets in the software product. It can either process the installation for all the filesets or invoke other executables for each fileset.

- Use the **inusave** command to save the current level of any files to be installed.
- Use **inurest** command to restore all required files for the usr part from the distribution media.
- Use the **inucp** command to copy all required files for the root part from the **/usr/lpp/***Package_Name***/inst_root** directory.
- Create an **$INUTEMPDIR/status** file indicating success or failure for each fileset being installed or updated. See “The Installation Status File” on page 604 for more information about this file.
- Return an exit code indicating the status of the installation. If the **instal** or **update** executable file returns a nonzero return code and no **status** file is found, the installation process assumes all filesets failed.

Optional Executable File Contained in the Fileset.al File

*Fileset***.unconfig_d**	Undoes fileset-specific configuration operations performed during the installation and updates of the fileset. The *Fileset***.unconfig_d** file is used when the **-u** flag is specified with the **installp** command. If this file is not provided and the **-u** flag is specified, the *Fileset***.unconfig**, *Fileset***.unpost_i**, and *Fileset***.unpre_i** operations are performed.

Further Description of Installation Control Files

The Fileset.cfgfiles File

The *Fileset***.cfgfiles** file lists configuration files that need to be saved in order to migrate to a new version of the fileset without losing user-configured data. To preserve user-configuration data, a *Fileset***.cfgfiles** file must be provided in the proper **liblpp.a** file (usr, root, or share).

The *Fileset***.cfgfiles** contains a one-line entry for each file to be saved. Each entry contains the file name (a path name relative to root), a white-space separator, and a keyword describing the handling method for the migration of the file. The handling method keywords are:

preserve	Replaces the installed new version of the file with the saved version from the save directory. After replacing the new file with the saved version, the saved file from the configuration save directory is deleted.
user_merge	Leaves the installed new version of the file on the system and keeps the old version of the file in the configuration save directory. The user will be able to reference the old version to perform any merge that may be necessary.
auto_merge	During the *Fileset***.post_i** processing, the product-provided executables merge necessary data from the installed new version of the file into the previous version of the file saved in the configuration save directory. After the *Fileset***.post_i** processing, the **installp** command replaces the installed new version of the file with the merged version in the configuration save directory (if it exists) and then removes the saved file.
hold_new	Replaces the installed new version of the file with the saved version from the save directory. The new version of the file is placed in the configuration save directory in place of the old version. The user will be able to reference the new version.

other	Used in any case where none of the other defined handling methods are sufficient. The **installp** command saves the file in the configuration save directory and provides no further support. Any other manipulation and handling of the configuration file must be done by the product-provided executables. The product developer has the responsibility of documenting the handling of the file.

The *Fileset*.**post_i** executable can be used to do specific manipulating or merging of configuration data that cannot be done through the the default installation processing.

Configuration files listed in the *Fileset*.**cfgfiles** file are saved in the configuration save directory with the same relative path name given in the *Fileset*.**cfgfiles** file. The name of the configuration save directory is stored in the **MIGSAVE** environment variable. The save directory corresponds to the part of the package being installed. The following directories are the configuration save directories:

/usr/lpp/save.config	For the usr part
/lpp/save.config	For the root part
/usr/share/lpp/save.config	For the share part

If the list of files that you need to save varies depending on the currently installed level of the fileset, the *Fileset*.**cfgfiles** file must contain the entire list of configuration files that might be found. If necessary, the *Fileset*.**post_i** executable (or executables provided by other products) must handle the difference.

For example, you have a fileset (**foo.rte**) that has one file that can be configured. So, in the root **foo.rte.cfgfiles**, there is one file listed:

```
/etc/foo_user   user_merge
```

When migrating from your old fileset (**foo.obj**) to **foo.rte**, you cannot preserve this file because the format has changed. However, when migrating from an older level **foo.rte** to a newer level **foo.rte**, the file can be preserved. In this case, you might want to create a **foo.rte.post_i** script that checks to see what fileset you are migrating from and acts appropriately. This way, if a user had made changes to the **/etc/foo_use**r file, they are saved.

The root **foo.bar.post_i** script could be as follows:

```
#! /bin/ksh
grep -q foo.rte $INSTALLED_LIST
if [$? = 0]
then
  mv $MIGSAVE/etc/foo_user/ /etc/foo_user
fi
```

$INSTALLED_LIST is created and exported by **installp**. See Installation for Control Files Specifically for Repackaged Products (“Installation Control Files Specifically for Repackaged Products” on page 592) for more information about the *Fileset*.**installed_list** configuration file. The *$MIGSAVE* variable contains the name of the directory in which the root part configuration files are saved.

The **installp** command does not produce a warning or error message if a file listed in the *Fileset*.**cfgfiles** file is not found. The **installp** command also does not produce a message for a file that is not found during the phase following *Fileset*.**post_i** processing when saved configuration files are processed according to their handling methods. If any warning or error messages are desired, the product-provided executables must generate the messages.

As an example of the *Fileset*.**cfgfiles** file, the `Product_X.option1` fileset must recover user configuration data from three configuration files located in the root part of the fileset. The `Product_X.option1.cfgfiles` is included in the root part of the `liblpp.a` file and contains the following:

```
./etc/cfg_leaf    preserve
./etc/cfg_pudding user_merge
./etc/cfg_newton  preserve
```

The Fileset.fixdata File

Fileset.**fixdata** — A stanza-format file that describes the fixes contained in the fileset update (or in a fileset installation, if used in place of an update)

The information in this file is added to a fix database. The **instfix** command uses this database to identify fixes installed on the system. If the *Fileset*.**fixdata** exists in a package, the fix information in the fix database is updated when the package is applied.

Each fix in the fileset should have its own stanza in the *Fileset*.**fixdata** file. A *Fileset*.**fixdata** stanza has the following format:

```
fix:
    name = FixKeyword
    abstract = Abstract
    type = {f | p}
    filesets = FilesetName FilesetLevel
    [FilesetName FilesetLevel ...]
    [symptom = [Symptom]]
```

FixKeyword can not exceed 16 characters. *Abstract* describes the fix and can not exceed 60 characters. In the **type** field, **f** represents a fix, and **p** represents a preventive maintenance update. The **filesets** field contains a new-line separated list of filesets and fileset levels. *FilesetLevel* is the initial level in which the fileset delivered all or part of the fix. *Symptom* is an optional description of the problem corrected by the fix. *Symptom* does not have a character limit.

The following example shows a *Fileset*.**fixdata** stanza for problem `MS21235`. The fix for this problem is contained in two filesets.

```
fix:
    name = MS21235
    abstract = 82 gigabyte diskette drive unusable on Mars
    type = f
    filesets = devices.mca.8d77.rte 12.3.6.13
            devices.mca.8efc.rte 12.1.0.2
    symptom = The 82 gigabyte subatomic diskettes fail to operate in a Martian environment.
```

The Fileset.inventory File

Fileset.**inventory** — File that contains specific information about each file that is to be installed or updated for the fileset

sysck — Command that uses the *Fileset*.**inventory** file to enter the file name, product name, type, checksum, size, link, and symlink information into the software information database

The *Fileset*.**inventory** file is required for each part of a fileset that installs or update files. If the package has a root part that does not contain files to be installed (it does configuration only), the root part does not require the *Fileset*.**inventory** file.

Note: The *Fileset*.**inventory** file does not support files which are greater than 2 gigabytes (>2GB) in size. If you ship a file that is greater than 2GB, include it in your *fileset*.**al** file, but not in your *Fileset*.**inventory** file. **sysck** has not been updated to handle files larger than 2GB, and the **/usr** file system on most machines will not be created with capability for files greater than 2GB (by default).

The inventory file consists of ASCII text in stanza format. The name of a stanza is the full path name of the file to be installed. The stanza name ends with a colon (:) and is followed by a new-line character. The file attributes follow the stanza name and have the format *Attribute*=V*alue*. Each attribute is described on a separate line. The following list describes the valid attributes of a file:

Attribute **Description**

class
The logical group of the file. A value must be specified because it cannot be computed. The value is ClassName [ClassName].

type
Specifies the file type. The **type** attribute can have the following values:

Type	Meaning
FILE	Ordinary file.
DIRECTORY	Directory.
SYMLINK	A symbolic link to a file.
FIFO	First-in-first-out file.
BLK_DEV	Block device special file.
CHAR_DEV	Character device special file.
MPX_DEV	Multiplexed device special file.

owner
Specifies the file owner. The attribute value can be in the owner name or owner ID format.

group
Specifies the file group. The attribute value can be in the group name or group ID format.

mode
Specifies the file mode. The value must contain the permissions of the file in octal format. Any of the following keywords can precede the permissions value. Items in the list are separated by commas.

Mode Items	Meaning
tcb	Part of the Trusted Computing Base.
tp	Part of the Trusted Process.
svtx	Text will be saved on swap for this file.
suid	File has the set user ID bit set.
sgid	File has the set group ID bit set.

target
Valid only for **type=SYMLINK**. The attribute value is the path name of the file to which the link points.

program
Specifies the software product to use to verify the file. This attribute is not usually used.

source	Specifies the location of the original copy of the file.
size	Specifies the size of the file in blocks. If the file size is expected to change through normal operation, the value for this attribute must be **VOLATILE**.
checksum	Specifies the checksum values of the file. The attribute value is a string containing the checksum value and number of 1024-byte blocks in the file as generated by the **sum** command. If the files size is expected to change through normal operation, the value for this attribute must be **VOLATILE**.
link	Specifies any hard links. If multiple hard links exist, each link is separated by a comma.

Note: The **sysck** command creates hard links and symbolic links during installation if those links do not exist. The root part symbolic links should be packaged in the root part *Fileset***.inventory** file.

Installation Control Files Specifically for Repackaged Products

The Fileset.installed_list File

*Fileset***.installed_list**	File created by the **installp** command when installing the fileset from a package if it is found that the fileset (or some form of it) is already installed on the system at some level

The software information database is searched to determine if either *Fileset* or any filesets listed in the file *Fileset***.namelist** (if it exists) are already installed on the system. If so, the fileset and the installation level are written to the *Fileset***.installed_list** file.

If it is created, the *Fileset***.installed_list** is available at the time of the calls to the **rminstal** and **instal** executables. The *Fileset***.installed_list** file can be located in the following directories, the packaging working directories, or *PackageWorkDirectory*:

/usr/lpp/	*PackageName* for the usr part
/lpp/	*PackageName* for the root part
/usr/share/lpp/	*PackageName* for the share part

The *Fileset***.installed_list** file contains a one-line entry for each fileset that was installed. Each entry contains the fileset name and the fileset level.

For example, while the `storm.rain 1.2.0.0` fileset is being installed, the **installp** command discovers that `storm.rain 1.1.0.0` is already installed. The **installp** command creates the *PackageWorkDirectory*`/storm.rain.installed_list` file with the following contents:

```
storm.rain 1.1.0.0
```

As another example, the `Baytown.com` fileset contains a `Baytown.com.namelist` file with the following entries:

```
Pelly.obj
GooseCreek.rte
CedarBayou.stream
```

While installing the Baytown.com 2.3.0.0 fileset, the **installp** command finds that Pelly.obj 1.2.3.0 and CedarBayou.stream 4.1.3.2 are installed. The **installp** command creates the*PackageWorkDirectory*/Baytown.com.installed_list file with the following contents:

```
Pelly.obj 1.2.3.0
CedarBayou.stream 4.1.3.2
```

The Fileset.namelist File

Fileset.**namelist**	File necessary when the fileset name has changed or the fileset contains files previously packaged in obsolete filesets. It contains names of all filesets that previously contained files currently included in the fileset to be installed. Each fileset name must appear on a separate line.

The *Fileset*.**namelist** file must be provided in the usr, root, or share part of the **liblpp.a** file. The *Fileset*.**namelist** file is only valid for installation packages; it is not valid for update packages.

At the beginning of installation, the **installp** command searches the Software Vital Product Data (SWVPD) to determine if the fileset or any fileset listed in the *Fileset*.**namelist** file is already installed on the system. The **installp** command writes to the *Fileset*.**installed_list** file the fileset names and fileset levels that are found installed, making this information available to product-provided executables.

As a simple example of a *Fileset*.**namelist** file, the small.business fileset replaces an earlier fileset named family.business. The small.business product package contains the small.business.namelist file in its usr part **liblpp.a** file. The small.business.namelist file contains the following entry:

```
family.business
```

As a more complex example of a *Fileset*.**namelist** file, a fileset is divided into a client fileset and a server fileset. The LawPractice.client and LawPractice.server filesets replace the earlier lawoffice.mgr fileset. The LawPractice.server fileset also contains a few files from the obsolete BusinessOffice.mgr fileset. The LawPractice.client.namelist file in the **liblpp.a** file for the LawPractice package contains the following entry:

```
lawoffice.mgr
```

The LawPractice.server.namelist file in the **liblpp.a** file for the LawPractice package contains the following entries:

```
lawoffice.mgr
BusinessOffice.mgr
```

If the *Fileset*.**namelist** file contains only one entry and the current fileset is not a direct replacement for the fileset listed in the *Fileset*.**namelist** file, you must include a *Fileset*.**rm_inv** file in the **liblpp.a** file. The installation process uses the *Fileset*.**namelist** file and the *Fileset*.**rm_inv** file to determine if a fileset is a direct replacement for another fileset. If the *Fileset*.**namelist** file contains only one entry and there is no *Fileset*.**rm_inv** file, the installation process assumes the new fileset is a direct replacement for the old fileset. When the new (replacement) fileset is installed, the installation process removes from the system all files from the old (replaced) fileset, even files not included in the new fileset.

In the previous examples, the small.business fileset is a direct replacement for the family.business fileset, so a small.business.rm_inv file should not exist. The LawPractice.client fileset is not a direct replacement for the lawoffice.mgr fileset, so a LawPractice.client.rm_inv file must exist, even if it is empty.

The Fileset.rm_inv File

Fileset.**rm_inv**	File that contains a list of files, links, and directories to be removed from the system if they are found installed

This file is used when the current fileset is packaged differently from a previous level of the fileset and the installation process should not remove previously installed files based on the fileset's entries in the inventory database.

A simple name change for a fileset is not sufficient to require a *Fileset*.**rm_inv** file. The *Fileset*.**rm_inv** file is necessary when a new fileset is either a subset of a previous fileset or a mixture of parts of previous filesets. If a *Fileset*.**namelist** file exists and contains entries for more than one fileset, you must use the *Fileset*.**rm_inv** file to remove previously installed levels of the files from the system.

The *Fileset*.**rm_inv** file consists of ASCII text in stanza format. The name of a stanza is the full path name of the file or directory to be removed if found on the system. The stanza name ends with a colon (:) and is followed by a new-line character. If attributes follow the stanza name, the attributes have the format *Attribute*=*Value*. Attributes are used to identify hard links and symbolic links that need to be removed. Each attribute is described on a separate line. The following list describes the valid attributes:

Attribute	Description
links	One or more hard links to the file. The full path names of the links are delimited by commas.
symlinks	One or more symbolic links to the file. The full path names of the links are delimited by commas.

For example, the `U.S.S.R 19.91.0.0` fileset contains the following files in the **/usr/lib** directory: `moscow`, `leningrad`, `kiev`, `odessa`, and `petrograd` (a symbolic link to `leningrad`). The product developers decide to split the `U.S.S.R 19.91.0.0` fileset into two filesets: `Ukraine.lib 19.94.0.0` and `Russia.lib 19.94.0.0`. The `Ukraine.lib` fileset contains the `kiev` and `odessa` files. The `Russia.lib` fileset contains the `moscow` file. The `leningrad` file no longer exists and is replaced by the `st.petersburg` file in the `Russia.lib` fileset.

The `Ukraine.lib.rm_inv` file must exist because the `Ukraine.lib` fileset is not a direct replacement for the `U.S.S.R` fileset. The `Ukraine.lib.rm_inv` file should be empty because no files need to be removed when the `Ukraine.lib` fileset is installed to clean up the migrating `U.S.S.R` fileset.

The `Russia.lib.rm_inv` file must exist because the `Russia.lib` fileset is not a direct replacement for the `U.S.S.R` fileset. If the `Russia.lib.rm_inv` file is used to remove the `leningrad` file when the `Russia.lib` fileset is installed, the `Russia.lib.rm_inv` file would contain the following stanza:

```
/usr/lib/leningrad:
     symlinks = /usr/lib/petrograd
```

Installation Files for Supplemental Disk Subsystems

A disk subsystem that will not configure with the provided SCSI or bus-attached device driver requires its own device driver and configuration methods. These installation files are provided on a supplemental diskette (which accompanies the device) and must be in backup format with a **./signature** file and a **./startup** file. The signature file must contain the string **target**. The startup file must use restore by name to extract the needed files from the supplemental diskette and to run the commands necessary to bring the device to the available state.

Format of Distribution Media

The following types of media can be used to distribute software product installation packages.

- "Tape"
- "CD-ROM"
- "Diskette" on page 596

The following sections describe the formats that must be used to distribute multiple product packages on each of these media.

Tape

In order to stack multiple product package images onto either a single tape or a set of tapes, the files on each tape in the set must conform to the following format:

- File 1 is empty. (Reserved for bootable tapes.)
- File 2 is empty. (Reserved for bootable tapes.)
- File 3 contains a table of contents file that describes product package images on the set of tapes. Therefore, each tape in the set contains a copy of the same table of contents file, except for the difference of the tape volume number in a multi-volume set. See "The Table of Contents File" on page 596 for more information.
- Files 4 through (*N*+3) contain the backup-format file images for product packages 1 through *N*.
- A product package image file cannot span across two tapes.
- Each file is followed by an end-of-file tape mark.

CD-ROM

A CD-ROM that is to contain multiple product package images must be compliant with the Rock Ridge Group Protocol. Product packages should be stored in the **/usr/sys/inst.images** directory, which must contain the following:

- The backup-format file images of the product packages.
- A table of contents file named **.toc** that describes the product package images in the directory. See "The Table of Contents File" on page 596 for more information.

A multiple volume CD-ROM is a CD-ROM that has an additional directory structure to define a set of CD-ROMs as a single installable unit.

A multiple volume CD-ROM must conform to the following rules:

- A **/usr/sys/mvCD** directory exists with the following contents:
  1. A table of contents file (.toc) that describes the product package images on all of the CD-ROMs of the set. Each volume of the CD-ROM must have the same **.toc** in **/usr/sys/mvCD**.
  2. An ASCII file named volume_id in which the first line consists of the decimal volume number of the CD in the set1.
  3. A symbolic link named **vol% *n***, where ***n*** is the decimal volume number of the of the CD in the set. The target of the symbolic link must be a relative path to the directory of product packages on that particular volume of the CD. The standard value for the symbol link is **../inst.images**.
- The table of contents file (**.toc**) in the **/usr/sys/mvCD** conforms to the standard table of contents format. The special characteristic of the multiple volume **.toc** is that the location of each product package image begins with the directory entry **vol% *n***, where ***n*** indicates the volume which contains that particular product package.

Example:

Fileset A is in file **A.bff** on volume 1. Fileset B is in file **B.bff** on volume 2. The field in the table of contents file in **/usr/sys/mvCD** containing the location of the product package images for A and B are **vol%1/A.bff** and **vol%2.bff**, respectively. The field in the table of contents file in **/usr/sys/inst.images** of volume 1 contains the location of A as **A.bff**. The field in the table of contents file in **/usr/sys/inst.images** of volume 2 contains the location of B as **B.bff**.

Note: Multiple volume CD-ROMs are not recognized on AIX systems prior to AIX 4.3. Each volume of the CD-ROM will be processed separately. The CD-ROMs should be produced whenever possible so that each volume may be processed separately, since there can be situations where a volume of the CD-ROM is unmountable and only the single volume may be accessible.

Diskette

In order to stack multiple product package images onto a set of diskettes, the following files must be written to the set of diskettes:

- A table of contents file that describes product package images to be included in the set. See "The Table of Contents File" for more information.
- Each product package image file that is to be included in the set.

The files are written to the set of diskettes using the following rules:

- Write the data as a stream to the diskette set with a volume ID sector inserted in block 0 of each diskette in the set. The data from the last block of one volume is treated as logically adjacent to the data from block 1 of the next volume (the volume ID sector is verified and discarded when read).
- Each file begins on a 512-byte block boundary.
- Write the table of contents file first. Pad this file to fill its last sector with null characters (x'00'). At least one null character is required to mark the end of the table of contents file. Thus, a full sector of null characters may be required.
- Following the table of contents file, write each of the product package image files to successive sectors. Pad each file to fill its last sector using null characters. A null character is not required if the file ends on the block boundary.
- Block 0 of each diskette in the set contains a volume ID sector. The format of this sector is:

Bytes 0:3	A magic number for identification. This is a binary integer with a value of decimal 3609823513=x'D7298918'.
Bytes 4:15	A date and time stamp (in ASCII) that serves as the identification for the set of diskettes. The format is *MonthDayHourMinuteSecondYear*. The *Hour* should be a value from 00 to 23. All date and time fields contain two digits. Thus, *Month* should be represented as 03 instead of 3, and *Year* should be represented as 94 instead of 1994.
Bytes 16:19	A binary integer volume number within this set of diskettes. The first diskette in the set is x'00000001'.
Bytes 20:511	Binary zeroes.

The Table of Contents File

The following table describes the table of contents file. Note that some fields are different for the different types of media.

The Table of Contents File			
Field Name	**Format**	**Separator**	**Description**

1. Volume	Character	White space	For the tape and diskette table of contents file, this is the number of the volume containing this data. For the fixed disk or CD-ROM table of contents file, the volume number is `0`.
2. Date and time stamp	*mmddhhMMssyy*	White space	For tape or diskette, this is the time stamp when volume 1 was created. For fixed disk or CD-ROM, this is the time stamp when the **.toc** file was created. See Date and Time Stamp Format later in this article for detailed description.
3. Header format	Character	New line	A number indicating the format of the table of contents file. Valid entries are: 1 -AIX Version 3.1, 2 -Version 3.2, 3 -AIX Version 4.1, B -**mksysb** tape (invalid for use by **installp**)
4. Location of product package image	Character	White space	For tape or diskette, this is a character string in the form: *vvv*.*bbbbb*:*sssssss*See Location Format for Tape and Diskette later in this article for detailed description. For fixed disk or CD-ROM, this is the file name of the product package image file. Note that this is the file name only and must not be preceded by any part of the path name.
5. Package specific information	**lpp_name** file format	New line	The contents of the **lpp_name** file contained in this product package image. See The lpp_name Package Information File for detailed description.

Note: Items 4 and 5 described in the preceding table are repeated for each product package image contained on the media.

Date and Time Stamp Format

A date and time stamp format is an ASCII string that has the following format:

MonthDayHourMinuteSecondYear

The *Hour* should be a value from 00 to 23. All date and time fields contain two digits. Thus, *Month* should be represented as `03` instead of `3`, and *Year* should be represented as `94` instead of `1994`.

Location Format for Tape and Diskette

The location has the format of *vvv*:*bbbbb*:*sssssss* where each letter represents a digit and has the following meaning:

For tape

vvv is the volume number of the tape.

bbbbb is the file number on the tape of the product package image.

sssssss
is the size of the file in bytes.

For diskette

vvv is the volume number of the diskette.

bbbbb is the block number on diskette where the product package image file begins.

sssssss
is the size of the file in bytes (including padding to the end of the block boundary).

The installp Processing of Product Packages

The major actions that can be taken with the **installp** command are:

Apply When a fileset in a product installation package is applied, it is installed on the system and it overwrites any pre-existing version of that fileset, therefore *committing* that version of the fileset on the system. The fileset may be removed if the user decides the fileset is no longer required.

When a fileset update is applied, the update is installed and information is saved (unless otherwise requested) so that the update can be removed later. Fileset updates that have been applied can be committed or rejected later.

Commit When a fileset update is committed, the information saved during the apply is removed from the system. Committing already applied software does not change the currently active version of a fileset.

Reject When an update is rejected, information saved during the apply is used to change the active version of the fileset to the version previous to the rejected update. The saved information is then removed from the system. The reject operation is valid only for updates. Many of the same steps in the reject operation are performed in a **cleanup** operation when a fileset or fileset update fails to complete installation.

Remove When a fileset is removed, the fileset and its updates are removed from the system independent of their state (applied, committed, or broken). The remove operation is valid only for the installation level of a fileset.

Executables provided within a product package can tailor processing for the apply, reject, and remove operations.

Reinstalling a fileset does not perform the same actions that removing and installing the same fileset do. The reinstall action (see **/usr/lib/instl/rminstal**) cleans up the current files from the previous or the same version, but does not run any of the **unconfig** or **unpre*** scripts. Therefore, do not assume that the **unconfig** script was run. The **.config** script should check the environment before assuming that the unconfig was completed.

For example, for the **ras.berry.rte** fileset, the config script adds a line to root's **crontab** file. Reinstalling the **ras.berry.rte** fileset results in two **crontab** entries, because the **unconfig** script was not run on reinstall (which removed the **crontab** entry). The config script should always remove the entry and then add it again.

Processing for the Apply Operation

This section describes the steps taken by the **installp** command when a fileset or fileset update is applied.

1. Restore the **lpp_name** product package information file for the package from the specified device.
2. Verify that the requested filesets exist on the installation medium.
3. Check the level of the requested filesets to ensure that they may be installed on the system.
4. Restore control files from the **liblpp.a** archive library file into the *package directory* (**/usr/lpp/***Package_Name* for **usr** or **usr/root** packages and **/usr/share/lpp/***Package_Name* for **share** packages. The control files specifically for the **root** portion of a **usr/root** package reside in **/usr/lpp/***Package_Name***/inst_root/liblpp.a**).
5. Check disk space requirements.
6. Check that necessary *requisites* (filesets required to be at certain levels to use or install another fileset) are already installed or are on the list to be installed.
7. If this is an instalation package rather than an update package, determine if there are license agreement requirements which must be satisfied in order to proceed with the installation.
8. If this is an installation package rather than a fileset update package, search the software vital product data (SWVPD) to see if *Fileset* (the fileset being installed) or any filesets listed in the *Fileset***.namelist** file are already installed on the system at any level. If *Fileset* is already installed, write the fileset name and installed level to the *Work_Directory/Fileset***.installed_list** file. If no level of *Fileset* is installed, then if any filesets listed in the *Fileset***.namelist** file are installed, list all those filesets and levels in the *Work_Directory/Fileset***.installed_list** file. *Work_Directory* is the same as the package directory with the exception of **root** parts, which use **/lpp/***Package_Name*.
9. If this is an installation package rather than a fileset update package, call the **/usr/lib/instl/rminstal** script to do the following for each fileset being installed (unless otherwise specified, files checked for existence must have been restored from the **liblpp.a** control file library):
   a. If *Fileset***.pre_rm** exists, execute *Fileset***.pre_rm** to perform required steps before removing any files from this version or an existing version of *Fileset*.
   b. If *Work_Directory/Fileset* **.installed_list** exists, move the existing files listed in *Fileset*.cfgfiles to the configuration file save directory (indicated by the **MIGSAVE** environment variable).
   c. If a version of *Fileset* is already installed, remove files and SWVPD information (except history) for *Fileset*.

ELSE
If `Work_Directory/Fileset.installed_list` exists,
If `Fileset.rm_inv` exists or `Fileset.namelist` contains more than one fileset or the only fileset listed in `Fileset.namelist` is `bos.obj`,
Remove files and SWVPD inventory information for files listed in the file `Fileset.rm_inv`.
Remove files and SWVPD inventory information for files listed in the file `Fileset.inventory`.
Remove other SWVPD information for any filesets listed in `Fileset.namelist` which no longer have any SWVPD inventory information.

ELSE
If `Work_Directory/Fileset.installed_list` exists and contains only one fileset and `Fileset.namelist` contained only one fileset,
Remove files and SWVPD information (except history) for that fileset.

   d. For each part of a product package (**usr** part only, **share** part only, or **usr** followed by **root**)

1) Set INUTREE (*U* for**usr**, *M* for **root**, and *S* for **share**) and INUTEMPDIR (name of created temporary working directory environment variables.
2) If an installation package:

If an **instal** control program exists in the package directory (not recommended)
Execute **./instal**.
ELSE
Execute the default script **/usr/lib/instl/instal**.
ELSE /* update package */
Set INUSAVEDIR environment variable.
If an **update** control program exists in the package directory (not recommended)
Execute **./update**.
ELSE
Execute the default script **/usr/lib/instl/update**.

3) If a **status** file has been successfully created by **instal** or **update**

Use **status** file to determine success/failure of each fileset.
ELSE
Assume all requested filesets in package failed to apply.

4) If apply operation was successful for a fileset

Update the Software Vital Product Data (SWVPD).
Register associated license agreement requirement if one exists.
ELSE
Run cleanup (the recommended default **/usr/lib/instl/cleanup** or the package-supplied **lpp.cleanup** from package directory) to clean up the failed filesets.

Processing of the Default install/update Script

The **instal** or **update** executable is invoked from **installp** with the first parameter being the device being used for the installation or update. The second parameter is the full path name to the file containing the list of filesets to be installed or updated, referred to below as **$FILESETLIST**. The default **instal** and **update** scripts are linked together; processing varies based on whether it is invoked as **instal** or **update.** The current directory is the package directory. A temporary directory INUTEMPDIR is created in **/tmp** to hold working files. The referenced files are described in Description of Installation Control Files (“Further Description of Installation Control Files” on page 588).

The flow within the default **instal** and **update** script is as follows:

1. Do the following for each fileset listed in the **$FILESETLIST**:
   a.

If update
Execute *Fileset***.pre_u** (pre_update) if it exists.
ELSE
Execute *Fileset***.pre_i** (pre_installation) if it exists.

   b. Build a master list of files to be restored from the package by appending *Fileset***.al** to the new file INUTEMPDIR**/master.al**.
   c. If update and files specified to be saved and **lpp.acf** (archive control file) exists,
      `Save off the library archive members being updated.`
   d. If processing is successful, append this fileset to the list to be installed in file **$FILESETLIST.new**.
2. If update and file saving specified, call **inusave** to save current versions of the files.

3. If processing root part,

Call **inucp** to copy files from apply list to root part.
ELSE
Call **inurest** to restore files from apply list for usr or share parts.

4. Do the following for each fileset listed in **$FILESETLIST.new** (failure in any step is recorded in the **status** file and processing for that fileset ends):
   a. Determine if this fileset is installed at the same or older level or if filesets listed in the *Fileset***.namelist** are installed. Export environment variables INSTALLED_LIST and MIGSAVE if such a condition (called a *migration*).
   b. If processing an update,

Invoke *Fileset***.post_u** if it exists.
ELSE
Invoke *Fileset***.post_i** if it exists.
If *Fileset***.cfgfiles** exists, then call **/usr/lib/instl/migrate_cfg** to handle processing of configuration files according to their specified handling method.

   c. Invoke **sysck** to add the information in the *Fileset***.inventory** file to the software vital product database (SWVPD).
   d. Invoke the **tcbck** command to add the trusted computing base information to the system if the *Fileset***.tcb** file exists and the trusted computing base attribute **tcb_enabled** is set in the **/usr/lib/objrepos/PdAt** ODM database.
   e. Invoke **errupdate** to add error templates if *Fileset***.err** exists.
   f. Invoke **trcupdate** to add trace report format templates if *Fileset***.trc** exists.
   g. If update or if *Work_Directory/Fileset***.installed_list** exists, invoke each *Fileset***.odmdel** and *Fileset***.*.odmdel** script to process ODM database deletion commands.
   h. Invoke **odmadd** on each existing *Fileset***.odmadd** and *Fileset***.*.odmadd** to add information to ODM databases.
   i. If update,

Invoke *Fileset***.config_u** (fileset configuration update) if it exists.
ELSE
Invoke *Fileset***.config** (fileset configuration) if it exists.

   j. Update the **status** file indicating successful processing for the fileset.
5. Link control files for needed for fileset removal into the package's **deinstl** directory for future use. These files include the following files that might be present in the package directory:

   lpp.deinstal , *Fileset***. al**, *Fileset***. inventory**, *Fileset***. pre_d**,*Fileset***. unpre_i**, *Fileset***. unpre_u**, *Fileset***. unpost_i**, *Fileset***. unpost_u**, *Fileset***. unodmadd**, *Fileset***. unconfig**, *Fileset***. unconfig_u**, $SAVEDIR/*Fileset***. *.rodmadd**, and SAVEDIR/*Fileset***. *.unodmadd**

Processing for the Reject and Cleanup Operations

This section describes the steps taken by the **installp** command when a fileset update is rejected or when a fileset or fileset update fails to complete installation. The default **cleanup** and **reject** scripts located in **/usr/lib/instl** are linked together. Their logic differs slightly depending on whether the script was invoked as **reject** or **cleanup**. For **usr**/**root** filesets or fileset updates, the **root** part is processed before the **usr** part.

1. If rejecting, check requisites to ensure that all dependent product updates are also rejected.
2. For each part of a package (i.e., **usr**, **root**, or **share**)
   a. Set INUTREE (*U* for **usr**, *M* for **root**, and *S* for **share**) and INUTEMPDIR environment variables.

b. If **reject** control file exists in current directory (INULIBDIR)

Invoke **./lpp.reject**
ELSE
Invoke the default script **/usr/lib/instl/reject**

3. Update the Software Vital Product Data.

The **reject** executable is invoked from **installp** with the first parameter being undefined and the second parameter being the full path name to the file containing the list of filesets (referred to below as **$FILESETLIST**) to be rejected for the update.

The following files are referenced by the default **cleanup** and **reject** script. They are described in detail in "Further Description of Installation Control Files" on page 588.

The flow within the default **cleanup** and **reject** script is as follows:

1. Do the following for each fileset listed in **$FILESETLIST**:
   a. If invoked as **cleanup**, then read the line in the *Package_Name***.s** status file to determine which step the installation failed on and skip ahead to the undo action for that step. A **cleanup** operation will only begin at the step where the the installation failed. For example, if the installation of a fileset failed in the *Fileset*.**post_i** script, then the cleanup operation for that fileset would begin at step (i) below, since there are no actions to undo from subsequent steps in the installation.
   b. Undo any configuration processing performed during the installation:

      If rejecting an update,

Invoke *Fileset***.unconfig_u** if it exists
ELSE
Invoke *Fileset***.unconfig** if it exists.

   c. Run any *Fileset***.*.unodmadd** and/or *Fileset***.unodmadd** files to remove Object Data Manager (ODM) entries added during the installation.
   d. Run any *Fileset***.*.rodmadd** and/or *Fileset***.rodmadd** exist to replace ODM entries deleted during the installation.
   e. Invoke **trcupdate** if *Fileset***.undo.trc** exists to undo any trace format template changes made during the installation.
   f. Invoke **errupdate** if *Fileset***.undo.err** exists to undo any error format template changes made during the installation.
   g. Invoke **tcbck** to delete the trusted computing base information to the system if the *Fileset***.tcb** file exists and the trusted computing base attribute **tcb_enabled** is set in the **/usr/lib/objrepos/PdAt** ODM database.
   h. Invoke **sysck** if *Fileset***.inventory** exists to undo changes to the software information database.
   i. Undo any post_installation processing performed during the installation:

      If update,

Invoke *Fileset***.unpost_u** if it exists
ELSE
Invoke *Fileset***.unpost_i** if it exists.

   j. Build a master apply list (called **master.al**) from *Fileset***.al** files.
   k. Add *Fileset* to **$FILESETLIST.new**.
2. Do the following if **$INUTEMPDIR/master.al** exists.
   a. Change directories to **/** (**root**).
   b. Remove all files in **master.al**.

3. Do the following while reading **$FILESETLIST.new**.
   a. Call **inurecv** to recover all saved files.
   b. If update,

Invoke *Fileset***.unpre_u** if it exists
ELSE
Invoke *Fileset***.unpre_i** if it exists.

   c. Delete the install/update control files.
4. Remove the *Package_Name***.s** status file.

Processing for the Remove Operation

This section describes the steps taken by the **installp** command when a fileset is removed. For **usr**/**root** filesets or fileset updates, the **root** part is processed before the **usr** part.

1. Check requisites to ensure that all dependent filesets are also removed.
2. For each part of a product package (i.e., **usr**, **root**, or **share**)
   a. Set INUTREE (U for **usr**, M for **root**, and S for **share**) and INUTEMPDIR (**installp** working directory generated in **/tmp**) environment variables.
   b. Change directory to **INULIBDIR**.
   c. If **deinstal** control file exists in current directory

Invoke **./lpp.deinstal**
ELSE
Invoke the default script **/usr/lib/instl/deinstal**.

3. Remove files belonging to the fileset from the file system.
4. Remove fileset entries from the SWVPD except for history data.
5. Deactivate license agreement requirement registration for the fileset.

The **deinstal** executable is invoked from **installp** with the first parameter being the full path name to the file containing the list of filesets to be removed, referred to below as **$FILESETLIST**.

The flow within the default **deinstal** script is as follows:

1. Do the following for each fileset listed in input file **$FILESETLIST**:
2. If *Fileset***.unconfig_d** exists

   Execute *Fileset***.unconfig_d** to remove all configuration changes, Object Data Manager (ODM) changes, and error and trace format changes, and to undo all operations performed in the post-installation and preinstallation scripts for all updates and the base level installation.
3. If *Fileset***.unconfig_d** does not exist,
   a. For each update for that fileset

Run any *Fileset***.unconfig_u** to undo any update configuration processing.
Run any *Fileset***.*.unodmadd** and/or *Fileset***.unodmadd** to delete Object DataManager (ODM) entries added during the update.
Run any *Fileset***.*.rodmadd** and/or *Fileset***.rodmadd** to add Object Data Manager (ODM) entries deleted during the update.
Run **errupdate** if *Fileset***.undo.err** exists to undo error log template changes.
Run **trcupdate** if *Fileset***.undo.trc** exists to undo trace report template changes.
Run any *Fileset***.unpost_u** to undo any post-installation customization.

b. For the fileset base installation level,

Run any *Fileset***.*.unodmadd** and/or *Fileset***.unodmadd** to delete Object Data Manager (ODM) entries added during the installation.
Run any *Fileset***.*.rodmadd** and/or *Fileset***.rodmadd** to add Object Data Manager (ODM) entries deleted during the installation.
Run **errupdate** if *Fileset***.undo.err** exists to undo error log template changes.
Run **trcupdate** if *Fileset***.undo.trc** exists to undo trace report template changes.
Run *Fileset***.unconfig_i** to undo any installation configuration processing.
Run *Fileset***.unpost_i** to undo any post-file installation customization.

4. Remove the files and software data information installed with the fileset.
5. If *Fileset***.unconfig_d** does not exist,
   a. For each update for that fileset

Run any *Fileset***.unpre_u** to undo any pre-file installation customization.

   b. For the fileset base installation level

Run any *Fileset***.unpre_i** to undo any pre-file installation customization.

6. Delete any empty directories associated with the fileset.

   Note: If an error is returned from some call during the execution of the **deinstal** executable, the error will be logged, but execution will continue. This is different from the other scripts because execution for that fileset is normally canceled once an error is encountered. However, once the removal of a fileset has begun, there is no recovery; therefore, removal becomes a best effort once an error is found.

The Installation Status File

$INUTEMPDIR/status	File that contains a one-line entry for each fileset that was to be installed or updated

The **installp** command uses this **status** file to determine appropriate processing. If you create installation scripts, your scripts should produce a **status** file that has the correct format. Each line in the **status** file has the format:

StatusCode Fileset

The following list describes the valid *StatusCode* values:

Status Code	Meaning
s	Success, update SWVPD
f	Failure, perform cleanup procedure
b	Bypass, failed, cleanup not needed
i	Requisite failure, cleanup not needed
v	**sysck** verification failed

The following example of a **status** file indicates to the **installp** command that the installations for the **tcp.client** and **tcp.server** filesets of **bos.net** package were successful and the installation for the **nfs.client** fileset was not successful.

```
s bos.net.tcp.client
s bos.net.tcp.server
f bos.net.nfs.client
```

Installation Commands Used During Installation and Update Processing

inucp	Copies files from the /usr/lpp/*Package_Name*/inst_root directory to the / (root) file tree when installing the root part.
inurecv	Recovers saved files for installation failure or software rejection (**installp -r**).
inurest	Restores files from the distribution medium onto the system using an apply list as input.
inusave	Saves all files specified by an apply list into the save directory belonging to the software product.
inuumsg	Issues messages from the inuumsg.cat message catalog file for the software product being installed.
ckprereq	Verifies compatibility of the software product with any dependencies using requisite information supplied in the lpp_name file and information about already installed products found in the SWVPD.
sysck	Checks the inventory information during installation and update procedures. The **sysck** command is in the **/usr/bin** directory. Other commands listed previously are in the **/usr/sbin** directory.

For examples of their use, refer to the default installation script, **/usr/lib/instl/instal**.

Chapter 21. Documentation Library Service

The Documentation Library Service provides an application that allows users to read and search HTML online documents. The Documentation Library Service was formerly known as the Documentation Search Service.

This chapter provides specific instructions for:

- Application developers who are including HTML documentation with their application and want to use the Documentation Library Service to provide reading, navigation, and search functions for their manuals.
- Anyone who wants to place a documents on a system and allow users to use the Documentation Library Service to read, navigate and search their documents.

The Documentation Library Service includes a search engine and the Documentation Library CGI programs. The Documentation Library CGI programs are stored in and run by a web server on a documentation server computer.

When the Documentation Library CGI program is called by an application, it displays the Documentation Library GUI in the user's browser. The user can then read, navigate through, or search the documents displayed in the interface.

When the user enters a text string in the search fields in the Documentation Library interface, the search string is then sent to the Library Service which conducts the search, generates a search results page, and then passes that page back to the user's browser.

The Documentation Library Service does not actually search through documents. Instead it searches compressed copies, called *indexes*, of documents. This greatly increases performance. In order to use the service, indexes must be created for documents. When the indexes are copied or installed on a system, the indexes must be registered with the library service so that the service knows their names and locations.

A default library GUI is provided. However, using customization features, you can customize the Library GUI to change things such as the title, text, graphics, and which documents are searched.

Note: This chapter contains commands that are longer than the width of the page. To make sure that long commands are completely visible, they are split up and displayed on multiple lines. This is an example of a long command line that has been split for viewing:

```
/usr/IMNSearch/cli/itecrix -s server
     -x index_name
     -p /usr/docsearch/indexes/index_name/data
```

When you see a command displayed like this, you *must* type it all on *one* command line, with a space between each part. The above command parts would be typed like the following:

```
/usr/IMNSearch/cli/itecrix -s server -x index_name -p /usr/docsearch/indexes/index_name/data
```

This chapter contains the following topics:

c. Section C: Calling the Documentation Library Service From Your Documentation (for all versions)

4. “Creating Indexes of your Documentation” on page 616
5. “Removing Indexes of your Documentation” on page 624
6. “Packaging your Application’s Documentation” on page 624

Language Support

Currently, the AIX 4.3 Documentation Library Service can only search documents that are written in supported languages and codesets. Refer to the Language Support Table for specific information.

For information on any changes in language support, make sure to read the README files that come with any updates to the Documentation Library Service.

Writing your HTML Documents

Currently, the Documentation Library Service supports **searching** HTML documents that are written using the languages and codesets listed in the Language Support Table. All documents in a single index must be written using the same language and codeset. Note that even though a document is written in a supported language, it cannot be searched unless it is written using the codeset of characters listed in the table. The last column in the table shows the characters that must be used as the last two characters of the index name for an index that contains that language. For example, if you are going to create an index named `doc456` and it is written in Spanish in the 8859-1 codeset, you would name it `doc456es`.

For more information on codesets and locales see Locale Overview in *AIX 5L Version 5.1 System Management Guide: Operating System and Devices*.

The Portable ASCII codeset of characters is included inside all other codesets. So you can include Portable ASCII characters in documents in all languages.

If you are creating a document in a codeset other than ISO8859-1, the Netscape browser may have a problem with displaying ampersand encoded characters that are equivalent to characters outside the Portable ASCII characters. These characters will not display correctly. For example, if you are using `&copy` for the copyright symbol, this is equal to a character value that is not in the Portable ASCII codeset. It may not display properly in any non-ISO8859-1 codeset.

HTML documents must be customized for use with the Documentation Library Service by including Search links in each document that will call the search form. These search links can be placed anywhere in the document. For example, they can be in the body of the text, in a header at the top of each page, in a navigation frame - anywhere where users are able to view them. See the next section for information on how the search link must be written.

Users may be using an ASCII browser to view the documentation. If it is likely that end users will be using an ASCII browser, the HTML documentation should be ASCII user-friendly. This includes techniques such as using an **ALT** attribute in the **<IMG>** tag for users unable to view images and **<NOFRAMES>** tags for users with browsers that are not frames capable. Consult HTML reference material for other techniques.

Insert a title tag in each document. Document titles should be meaningful and unique. The document title will appear in the list of matched documents in the search results page as the title of the found document. The text between the **<TITLE>** and **</TITLE>** tags should contain the title of the document and no other HTML tags. Additionally, titles should have a maximum length of 256 bytes.

Making your Documents Printable

Books within the documentation library are structured as a collection of articles or chapters. These articles are loaded into the client web browser one at a time for reading. While this allows great flexibility in navigating through articles, it is difficult to print an entire book in one print action. Beginning in AIX 5.1, the Documentation Library Service contains a **Print Tool** button. When you clicks this button, list of books that can be downloaded in a single printable file is displayed. You have the option of including your book in this list for printing.

To have your book appear in this list, complete the following:

1. Create a single file that merges all of your book's files into a single file in a printable format. The format you use is up to you. AIX manual print files are in PDF format, but any format can be used.
2. Add the print file to your install package. The print file must be installed to be accessible from the machine's web server's document directory. By default, the **/usr/share/man/info** directory is always linked under the web server's document directory, therefore, it is recommended that you install the printable file for your book into **/usr/share/man/info**. For example, if your product is called *Esther*, and your book's print file is named *userguide.pdf*, you could install the print file as **/usr/share/man/info/esther/userguide.pdf**.

   Note: If you have different language versions of your book, you will need to ship and install a separate print file for each language. For example, if you have an English and a Spanish version of your book, you might install two printable files as follows:
   /usr/share/man/info/esther/english/userguide.pdf and
   /usr/share/man/info/esther/spanish/userguide.pdf
3. In your View Definition File (VDF) , include a **Printfile** tag on the entry line that defines the book to define the name and location of the file that contains the printable version of the book. The library service uses the path in this tag to create a link to your printable book in the printable books list in the Print Tool. During configuration, the library service automatically creates a link in the machine's web server's documentation directory which points to **/usr/share/man/info**. This link is named **doc_link**. Therefore, if you are installing your print file under the **/usr/share/man/info** directory, you must use the name **doc_link** instead of **/usr/share/man/info** as the first part of your path in the **Printfile** tag. Using our above English-only example, you would include the following tag in the entry line for your book in your VDF: `Printfile:/doc_link/esther/userguide.pdf`

For our two language example, you would use the following **Printfile** tags - the first one in the English VDF and the second in the Spanish VDF:

```
Printfile:/doc_link/esther/english/userguide.pdf
Printfile:/doc_link/esther/spanish/userguide.pdf
```

For more details on View Definition Files and the **Printfile** tag, see 612.

Calling the Documentation Library Service From Your Documentation

Navigation Strategies

Users can navigate your documents in two ways:

- **Global View Set** - Navigate all documents registered into the Global view set.

  To enable users to navigate and search your documentation in the Global view set, you must:

  1. Register your documentation into the Global view set's **Books** view.
     a. Create a View Definition File that describes the hierarchical structure of your documentation.
     b. Register the Contents of each of your View Definition Files.

Note: You may register your documentation into any of the other views of the Global view set. The process is the same for the **Books** view though a separate view definition file may be required for each view.

2. Create links in your documentation to the Global view set for your document's language.

   If your documentation is in English, the HTML link might be

   ```
   <A HREF="/cgi-bin/ds_form?lang=en_US">Home/Search</A>
   ```

- **Custom View Set** - Only navigate and search documents for your application

  To enable users to navigate and search your documentation separately from all other documentation on the system, you must create a Custom View Set.

Creating a Custom View Set

1. Create a View Set Configuration File
2. Create a View Definition File
3. Register the Contents of each of your View Definition Files
4. Create Links in your Documentation to your Custom View Set

1. **Create a View Set Configuration File**

   a. Create a view set directory. This directory is named **/usr/docsearch/views/***locale***/***view_set_name* where *locale* is the name of the language locale in which the documentation was written, and *view_set_name* is a name which uniquely identifies your view set.

   b. Create a view set configuration file named `config` in the view set directory.

   Example: If your view set is named `MyDocuments` and you want to create a set of English views, your configuration file should be in the location `/usr/docsearch/views/en_US/MyDocuments/config`

   Note: Lines in the configuration file beginning with a `#` are assumed to be comments and are ignored.

   There are many things that can be customized in the view set configuration file:

Field	Description and Examples
View Name and Label	The name of an view and the text to be displayed on the tabs which allow the user to change between different views of a view set. A separate view name is necessary because a view label may be different in other languages. For example, if you have a view named `Books` which you link to from your documentation, the name of the view will always be `Books` but in Spanish the label of the view could be `Libros`. `view =` *View_Name* <TAB> *View Label* **Example:** If the name of the view is `Tasks`, but you want the label on the tab to be `How To`, you would add the following line to the view set configuration file: `view = Tasks  How To`
title	The text to be displayed at the top of the library GUI and the browser window. `title =` *Library GUI Title* **Example:** If you want the text at the top of the library GUI to be `My Documentation`, you would add the following line to the view set configuration file: `title = My Documentation`

Field	Description and Examples

results_title	The text to be displayed at the top of the results GUI and the browser window containing the results GUI.
	
	`results_title = Results GUI Title`
	
	Example: If you want the text at the top of the library GUI to be `My Documentation Search Results`, you would add the following line to the view set configuration file:
	`results_title = My Documentation Search Results`

page_top	Replaces the default HTML header of the library GUI with the HTML code between the **page_top_begin** and **page_top_end** tags.
	
	Example: If you want the top of your library GUI to contain an image named `myimage.gif` which is in the web server's `/myimages` directory, and the title of browser window to be `My Documents`, you might insert the following in your view set configuration file:
	`page_top_begin`
	`<HTML>`
	`<HEAD>`
	`<TITLE>My Documents</TITLE>`
	`<BODY>`
	`<DIV ALIGN="CENTER">`
	`<IMG SRC="/myimages/myimage.gif">`
	`</DIV>`
	`<P>`
	`page_top_end`
	
	Note : If a **title** configuration file entry is specified, it will be ignored.

page_bottom	Replaces the default HTML footer of the library GUI with the HTML code between the **page_bottom_begin** and **page_bottom_end** tags.
	
	Example: If you want the bottom of the library GUI to have a **MAILTO** link so that users can send mail to you, you might insert the following in your configuration file:
	`page_bottom_begin`
	`<HR>`
	`<DIV ALIGN="CENTER">`
	`<A HREF="MAILTO:me@my.site">Feedback</A>`
	`</DIV>`
	`</BODY>`
	`</HTML>`
	`page_bottom_end`

results_top	Replaces the default HTML header of the results GUI with the HTML code between the **results_top_begin** and **results_top_end** tags.
	
	Example: If you want the top of your results page to contain an image named `myimage.gif` which is in the web server's `/myimages` directory, and the title of the browser window to be `Results of My Search`, you might insert the following in your view set configuration file:
	`results_top_begin`
	`<HTML>`
	`<HEAD>`
	`<TITLE>Results of My Search</TITLE>`
	`<BODY>`
	`<DIV ALIGN="CENTER">`
	`<IMG SRC="/myimages/myimage.gif">`
	`</DIV>`
	`results_top_end`
	
	Note: If a **results_title** configuration file entry was specified, it will be ignored.

Field	Description and Examples

results_bottom	Replaces the default HTML footer of the results GUI with the HTML code between the **results_bottom_begin** and **results_bottom_end tags**.
	
	Example: If you want the bottom of the results page to have a **MAILTO** link so that users can send mail to you, you might insert the following in your configuration file:
	`results_bottom_begin`
	`<HR>`
	`<DIV ALIGN="CENTER">`
	`<A HREF="MAILTO:me@my.site">Feedback</A>`
	`</DIV>`
	`</BODY>`
	`</HTML>`
	`results_bottom_end`

2. **Create a View Definition File**

   Create a view definition file for each view in your view set. The format of this file is as follows:

   ```
   #<TAB>Entry Title[<TAB>Field:Value...]
   #<TAB>Entry Title[<TAB>Field:Value...]
   #<TAB>Entry Title[<TAB>Field:Value...]
   ```

   where the # is the level of the entry in the hierarchical tree structure, the entry title is the text to be displayed in the library GUI, and the possible fields are those listed below. The first entry level in a view definition file is **always** 0 and can increase up to 9 as the depth of the entry increases. Entries with the same level # will be displayed with the same indentation. (You can think of the entry level as the number of times to indent the tree at that entry.)

Field	Description and Examples

Printfile	Beginning in AIX 5.1, this optional VDF tag is used to create a link to your book in the list of printable books in the Print Tool . When a user clicks on this link in the Print Tool, the library service will download a printable file containing your entire book to the user's browser. In addition to inserting this tag in your VDF, you will need to create and package the printable book file. For more information on these tasks, see "Making your Documents Printable" on page 609.
	
	The Printfile tag has the following syntax:
	`Printfile[1-20]:/doc_link/$path`
	
	This syntax displays the book in the list of printable books in the Print Tool. *$path* is the path to your printable file. This path must be a sub-directory of the **/usr/share/man/info** directory. The link **doc_link** is automatically placed by the library service in the machine's web server document directory. This link points the web server to the **/usr/share/man/info** directory and allows the web server to find your print file if it is installed under **/usr/share/man/info**.
	
	The number 1-20 is optional. It is used only when your book is too large to be contained in one printable file. You can split your book into as many as 20 different printable files for downloading. The number specifies which section of the book is located in the file defined by the path. A separate link will appear in the list of printable books for each section or **Printfile** tag for a book. If your book only has one file for downloading, omit the number entry.
	
	Examples: Assuming your application is called the Esther tool, and your book is called userguide.pdf, you could use the following tags:
	
	If you ship the entire book in one file, in only one language (English), use:
	`Printfile:/doc_link/esther/userguide.pdf`
	
	If you split the book up into two files for download, use:
	`Printfile1:/doc_link/esther/userguide_section1.pdf`
	`Printfile2:/doc_link/esther/userguide_section2.pdf`
	
	You ship an English version and a Spanish version. The first tag is used in your English VDF and the second is used in your Spanish VDF:
	`Printfile:/doc_link/english/esther/myownbigbook/userguide.pdf`
	`Printfile:/doc_link/spanish/esther/myownbigbook/userguide.pdf`

Checked	Specifies default search state (selected for search or not selected for search) (This applies only to custom views. You can not specify a default search state for the Global Views.) The value can be either `Yes` or `No`. If no **Checked** field is present for an entry, the default search state is for the entry to be selected for search (i.e. `Yes`). **Examples:Checked:** `Yes` **Checked:** `No`
Collate	Specifies whether to sort the children of this entry (whether entries directly under this entry are to be kept in the order given, or sorted lexicographically according to the locale). The value can be either `Yes` or `No`. If no **Collate** field is present for an entry, the default ordering is the order given (i.e. `No`). **Examples:Collate:** `Yes` **Collate:** `No`

Field	Description and Examples
Expand	Specifies whether this node of the tree is expanded or collapsed by default. (This applies only to custom views. You can not specify a default expansion state for the Global Views.) The value can be either `Yes` or `No`. If no **Expand** field is present for an entry, the default expansion state is the for the entry to be collapsed (i.e. `No`). **Examples:Expand:** `Yes` **Expand:** `No`
Extra	Specifies text that is to be displayed after the title, but that should not be a link when a URL is given. **Example: Extra**: Some other text that isn't part of the link
Icon	Specifies the filename of the icon which is to be displayed before the entry's title. Following is a list of icon's which are provided by the Documentation Library Service. • bookcase.gif • bookshelf.gif • book.gif • chapter.gif • paper.gif These icons reside in the directory **/usr/docsearch/images**. If you want to use your own icon, place it in that directory on the documentation server machine. If no **Icon** field is present for an entry, no icon will be displayed. **Example:** `Icon:book.gif` **Note**: Icons are assumed to be 24 pixels wide and 24 pixels high. If you use an icon which is larger or smaller than this size, the icon will be resized to 24 by 24.
Index	Name of the search engine index(es) of documents represented by this entry and its descendants. Multiple indexes can be specified by listing them, separated by commas. Once a view definition file specifies an index for an entry, no other view definition files will be able to add entries under that entry. This helps to ensure that the contents of the tree below the index are exactly the documents which were indexed. **Examples:** `Index:BSADMNEN` `Index: CMDS01EN,CMDS02EN,CMDS03EN, CMDS04EN,CMDS05EN,CMDS06EN`
Position	Suggested relative position within a container. For example, if you are inserting an article under a book and you want that article to appear as the third article in the book, you could assign it a position number of `3` if there were already articles with positions of `1` and `2`. In case of multiple entries with the same position, order will be determined by the value of **Collate** field of the parent entry. The position of an entry in the view definition file overrides any position field. Therefore, it is not necessary to specify positions for entries below the point at which no other books will be occupying the same space. If no **Position** field is present for an entry, the default position value is zero (`0`). **Example:** `Position:5`
URL	The URL of the document to go to for navigation. This is used to locate the document when a web server is being used. This value must be an absolute path, but must not contain the protocol (http://) or the name or port number of the web server. If no URL is specified, the entry will not be a HTML link when displayed. **Example:**`URL:/doc_link/en_US/a_doc_lib/cmds/aixcmds2/grep.htm`

Field	Description and Examples
Version	The version of this entry. When registering documentation, a higher numbered version on an entry will replace a previous lower version number. **Example:** `Version:4.3.2.0`

A portion of an example view definition file is below:

```
0  AIX Base Library        Position:1      Icon:library.gif
1  AIX System Management Guides    Position:1      Icon:bookshelf.gif
2  Operating System and Devices    Index:BADMNEN   Postion:1       Icon:book.gif
3  Chapter 1. System Management with AIX   URL:/doc_link/en_US/a_doc_lib/aixbman/baseadmn/Ch1.htm
    Icon:chapter.gif
4  The System Administrator's Objectives   URL:/doc_link/en_US/a_doc_lib/aixbman/baseadmn/Ch1.htm#CE13340208vick
    Icon:paper.gif
3  Chapter 2. Starting and Stopping the System     URL:/doc_link/en_US/a_doc_lib/aixbman/baseadmn/Ch2.htm
    Icon:chapter.gif
4  Starting the System     URL:/doc_link/en_US/a_doc_lib/aixbman/baseadmn/starting.htm
    Icon:paper.gif
4  Understanding the Boot Process  URL:/doc_link/en_US/a_doc_lib/aixbman/baseadmn/under_boot.htm Icon:paper.gif
        .
        .
        .
```

3. **Register the Contents of each of your View Definition Files**

   /usr/sbin/ds_reg **[-d]** *locale View_Set View view_definition_file*

   where *locale* is the locale (language) in which your documentation is written, *View_Set* is the name of the view set, *View* is the name of the view into which you wish to register your documentation, and *view_definition_file* is the location of the view definition file. The optional **-d** flag is used to unregister the contents of a registered view definition file.

   Example: If you have a view definition file in `/MyDocuments/Books.vdf`, and you want to register it into the English Global Books view, you would type the command:

   ```
   /usr/sbin/ds_reg en_US Global Books /MyDocuments/Books.vdf
   ```

   Example: If you have a view definition file in `/MyDocuments/Commands.vdf`, and you want to unregister it from the Spanish AIX Commands view, you would type the command:

   ```
   /usr/sbin/ds_reg -d es_ES AIX Commands /MyDocuments/Commands.vdf
   ```

4. **Create Links in your Documentation to your Custom View Set**

   The base URL of the Documentation Service CGI program is always **/cgi-bin/ds_form**. This URL can be modified by any of the following arguments. Multiple arguments are separated by an ampersand (`&`).

Argument	Description and Example
lang	The locale of the documentation you want to display. If no locale is specified the default locale of the documentation server will be used. **Example:** If you want to see Japanese documentation, your link might be `<A HREF="/cgi-bin/ds_form?lang=Ja_JP">`
viewset	The name of the view set you want to display. If no viewset is specified, the Global view set will be used. **Example:** If your view set is called `MyDocuments` your link might be `<A HREF="/cgi-bin/ds_form?viewset=MyDocuments">`
view	If no view is specified, the first view in the viewset will be used. **Example:** If you want to see the Commands view of the default (Global) view set, your link might be `<A HREF="/cgi-bin/ds_form?view=Commands">`

Argument	Description and Example
advanced	This argument specifies that you want to see the advanced search form. If no advanced argument is given, the simple search form will be displayed. **Example:** If you want to see the advanced version of the library GUI `<A HREF="/cgi-bin/ds_form?advanced">`

Example: If you want to create a link in your Spanish (`es_ES`) documentation to the Subroutines view of your Custom View Set `MyDocuments`, your link could be

```
<A HREF="/cgi-bin/ds_form?viewset=MyDocuments&view=Subroutines&lang=es_ES">
```

Creating Indexes of your Documentation

The search engine does not search your actual documentation files. Instead it searches indexes that are created from your documentation. Very simplistically, indexes are compressed copies of your files. This greatly speeds up the searches. Therefore, if you want to use the search service to search your documents, you must create at least one index that will be installed with your documents.

Requirements

Before beginning to create your indexes, make sure you meet the following requirements:

- If it is not already installed, install the Documentation Library Service package onto your development computer. For more information on installation and configuration, see “Chapter 21. Documentation Library Service” on page 607 in *System Management Guide: Operating System and Devices*.
- If it is not already installed, install the search engine authoring tools package - IMNSearch Build Time package (**IMNSearch.bld**) on your computer. This software is contained on the AIX Base Operating System media.
- To use the index creation tool, you must be a member of the **imnadm** index administrators user group. If your username is not a member of this group, have your system administrator add your user ID to this group. Or log in using another username that is a member of this group.

Building the Indexes

1. Choose a Unique Index Name
2. Create a New Directory
3. Create an ASCII File
4. Choose a Title for your Index
5. Create an Empty Index
6. Add your Documents to the Update List
7. Start the Index Updating Process to Build your Index
8. Update the Registration Table
9. Copy your HTML Documents from the Build Directory into the Documentation Directory
10. Test your Index
11. Final Step

Each index you create will have its own selection checkbox in the search form. Typically, you create one index that contains text from multiple documents. Each time that index is selected for search, all the documents in that index will be searched. So when you combine documents into an index, you should think about what documents your user will want to search together.

Also, if you are creating an installp package, all documents that are within one index should be placed inside the same installable unit (fileset) of documentation. Otherwise users might only install some of the documents within the index and they would get `missing document` errors when they try to open the documents from the search results page.

For each index you want to create, repeat the following steps:

1. **Choose a Unique Index Name**

   When you create a search index for a document you must specify an eight (8) character name for the index. However, the search service will not let you register your new index if there is already a registered index that has the same name as your index. To reduce the probability of naming conflicts, it is recommended that certain naming conventions be followed:

   - If you are not an application developer and are just creating indexes for documents written at your site, use `999` as the first three characters of all your index names. The middle three characters of the name can be any combination of letters and numbers. The last two characters of the name must specify the language and codeset of the document. The language is specified using the appropriate two character suffix listed in the Language Support Table.

     Example: If you are creating an index for a document you wrote in English and the ISO8859-1 codeset, the index name must end in `en`. You could name the index `999ak2en`.

   - If you are an application developer and you are creating indexes to package in your installp package, all of your index names should star with three characters that represent your application's name. The middle three characters of the name can be any combination of letters and numbers. The last two characters of the name must specify the language and codeset of the document. The language is specified using the appropriate two character suffix listed in the Language Support Table.

     Example: If your application is called `Calculator`, and the document you are indexing is written in English and the ISO8859-1 codeset, the index name must end in `en`. You could name the index `cal2b4en`.

   - The following table shows the required index name endings (suffixes) for each supported language/codeset combination.

     Language Support Table

Language	Codeset	Locale	Index Name Suffix	Support Started in AIX:
Catalan	ISO8859-1	ca_ES	*name* ca	4.3.0
	ISO8859-15	ca _ES.8859-15	*name* c5	4.3.2
Danish	ISO8859-1	da_DK	*name* da	4.3.0
Dutch Netherlands	ISO8859-1	nl_NL	*name* nl	4.3.0
	ISO8859-15	nl_NL.8859-15	*name* b5	4.3.2
English United States	ISO8859-1	en_US	*name* en	4.3.0
	ISO8859-1	C	*name* en	4.3.0
English Great Britain	ISO8859-1	en_GB	*name* gb	4.3.0
Finnish	ISO8859-1	fi_FI	*name* fi	4.3.0
	ISO8859-15	fi_FI.8859-15	*name* u5	4.3.2
French	ISO8859-1	fr_FR	*name* fr	4.3.0
	ISO8859-15	fr_FR.8859-15	*name* f5	4.3.2
French Canada	ISO8859-1	fr_CA	*name* fc	4.3.0
German	ISO8859-1	de_DE	*name* de	4.3.0
	ISO8859-15	de_DE.8859-15	*name* d5	4.3.2

Language	Codeset	Locale	Index Name Suffix	Support Started in AIX:
German Switzerland	ISO8859-1	de_CH	*name* cd	4.3.0
Icelandic	ISO8859-1	is_IS	*name* is	4.3.0
Italian	ISO8859-1	it_IT	*name* it	4.3.0
	ISO8859-15	it_IT.8859-15	*name* i5	4.3.2
Norwegian	ISO8859-1	no_NO	*name* no	4.3.0
Portuguese, Brazilian	ISO8859-1	pt_BR	*name* pt	4.3.0
Portuguese, Portugal	ISO8859-1	pt_PT	*name* po	4.3.0
	ISO8859-15	pt_PT.8859-15	*name* y5	4.3.2
Russian	ISO8859-5	ru_RU	*name* ru	5.0.0
Spanish	ISO8859-1	es_ES	*name* es	4.3.0
	ISO8859-15	es_ES.8859-15	*name* s5	4.3.2
Swedish	ISO8859-1	sv_SE	*name* sv	4.3.0
Japanese	IBM-932	Ja_JP	*name* jp	4.3.2
Korean	IBM-eucKR	ko_KR	*name* kr	4.3.2
Simplified Chinese	IBM-eucCN	zh_CN	*name* cn	4.3.2
Traditional Chinese	big5	Zh_TW	*name* tw	4.3.2

2. **Create a New Directory**

   Create a new directory to hold the documents that will go into the index. We will call this directory the **build** directory. The build directory can be any place you want it. In our examples we are building indexes for a calculator application, so our build directory will be named `/usr/work/calculator`. Inside this build directory, arrange the documents into a directory tree structure exactly as you want them to be installed/placed relative to each other on a documentation search server computer.

   The result is that each document will have a full pathname that is composed of a “temporary” part, and a “permanent” part. The temporary part is the pathname of the build directory. The permanent part of the path specifies the location of the document inside your document tree. Once an index is built, the permanent part of a document’s pathname cannot be changed. The one rule about the pathnames is that the first directory in the permanent part of the pathname must be the index name.

   For example, your application is called `calculator`. The online documents for the application are written in US English. There are two user guide documents (doc1, doc2) and one administrator document (doc3). You could place the documents like this in the filesystem on the computer on which you are building the indexes:

   ```
   /usr/work/calculator/user/doc1.html
   /usr/work/calculator/user/doc2.html
   /usr/work/calculator/admin/doc3.html
   ```

   You can place your build directory anywhere, but all documents that go into a single index must under a single directory which acts as the common top directory so that they form a single tree. In the example, `calculator` is the common top directory.

3. **Create an ASCII File**

   For each index, create a document list file. Place inside this file a list of all the documents you want to be in the index. For each document, list it by using the full pathname that specifies where the document can be currently found on your development computer. Note that the working locations of these documents do not need to be the same location where the documents will be eventually installed on a documentation server. This document list file can be named anything and placed in any directory. Put each pathname on its own line in the file.

If you arranged your documents like the example above, your ASCII file would have the following contents:

```
/usr/work/calculator/user/doc1.html
/usr/work/calculator/user/doc2.html
/usr/work/calculator/admin/doc3.html
```

Next you must indicate where the temporary part of each pathname ends and where the permanent "installed" part of the pathnames start. You do this by replacing the last slash (/) in the temporary part of the document pathnames (the build directory pathname) with a commercial at symbol (@). When the index is created, only the part of each pathname that is to the right of the @ will be saved in the index.

For example, the above example file would now be modified to look like this:

```
/usr/work@calculator/user/doc1.html
/usr/work@calculator/user/doc2.html
/usr/work@calculator/admin/doc3.html
```

The slash before the application name (calculator) was replaced with an @ since it is the last slash in the temporary part of the path.

4. **Choose a Title for your Index**

   The title of your index is the text that will appear next to the index's checkbox in the search form. The title should uniquely describe the document or documents that are in the index and contain a maximum of 150 characters.

5. **Create an Empty Index**

   You must then create an empty index. After the index is created you will fill it. To prepare for index creation, you must check the following:

   Your user id must be a member of the imnadm group to use the steps that follow. Before you can create your first index you will need to change ownership of the /usr/docsearch/indexes directory so that it is owned by the user imnadm. You will only need to do this step before you create your first index.

   chown imnadm:imnadm /usr/docsearch/indexes

   Creation of an index requires three steps.

   a. Create the empty index.

      The index creation command has the syntax (all 5 lines on one command line):

      ```
      /usr/IMNSearch/bin/itecrix -s server -x index_name
      -p /usr/docsearch/indexes/index_name/data
      -pw /usr/docsearch/indexes/index_name/work
      -lsse itelsswt
      -t NORM | -t NGRAM
      -ccsid <codeset_id>
      ```

      Where *index_name* is the 8 character name of the index. The values for -t and -ccsid depend on the language of the documents in the index. Note that all single-byte languages have a -t value of NORM. All multi-byte languages have a -t value of NGRAM and you also must add the -ccsid value when creating a multi-byte index. The following table specifies the values to use for each language:

Language	-t	-ccsid	-lang
English (United States) ISO8859-1	NORM	819	EN_US
English (United States) ISO8859-15	NORM	923	EN_US
English Great Britain ISO8859-1	NORM	819	EN_GB
Catalan ISO8859-1	NORM	819	CA_ES
Catalan ISO8859-15	NORM	923	CA_ES

French ISO8859-1	NORM	819	FR_FR
French ISO8859-15	NORM	923	FR_FR
French Canadian ISO8859-1	NORM	819	FR_FR
German ISO8859-1	NORM	819	DE_DE
German ISO8859-15	NORM	923	DE_DE
German Switzerland ISO8859-1	NORM	819	DE_CH
Icelandic ISO8859-1	NORM	819	IS_IS
Italian ISO8859-1	NORM	819	IT_IT
Italian ISO8859-15	NORM	923	IT_IT
Norwegian ISO8859-1	NORM	819	NO_NO
Portuguese, Brazil ISO8859-1	NORM	819	PT_BR
Portuguese, Portugal ISO8859-1	NORM	819	PT_PT
Portuguese, Portugal ISO8859-15	NORM	923	PT_PT
Russian ISO8859-9	NORM	878	RU_RU
Spanish ISO8859-1	NORM	819	ES_ES
Spanish ISO8859-15	NORM	923	ES_ES
Swedish ISO8859-1	NORM	819	SV_SE
Japanese IBM-932	NGRAM	932	JA_JP
Korean IBM-eucKR	NGRAM	949	KO_KR
Traditional. Chinese big5	NGRAM	950	ZH_TW
Simplified Chinese IBM-eucCN	NGRAM	1381	ZH_CN

Following our example, to create a single byte English index, you could type (all 5 lines on one command line):

```
/usr/IMNSearch/bin/itecrix -s server -x cal413en
-p /usr/docsearch/indexes/cal413en/data
-pw /usr/docsearch/indexes/cal413en/work
-lsse itelsswt
-t NORM
```

b. Next you must specify the language and codeset of the documents that will be inserted into the index. The language specfication command has the following format (all on one line):

```
/usr/IMNSearch/bin/iterulix -s server -x index_name -dfmt HTML
-ccsid <codeset_id
-lang <language
```

where *index_name* is the same name that was used in the previous command and *codeset_id* and *language* are the values from the table above.

Following our example, you would now type (all on one line):

```
/usr/IMNSearch/bin/iterulix -s server -x cal413en -dfmt HTML -ccsid 819
-lang EN_US
```

c. After you create your index, you should check to make sure that your index is listed with the Documentation Library Service by typing:

/usr/IMNSearch/bin/itelstix -s server

6. **Add your Documents to the Update List**

Next you must tell the Documentation Library Service the name of the file that contains the list of the documents that will go into the the empty index you just created. Then later you will run an update command and those documents will be indexed and the results will be inserted in the index.

Use the following command to add your documents to the list of documents that will get inserted into the index

```
/usr/IMNSearch/bin/itequeue -s server -x index_name -add -l document_list_file
```

(where *document_list_file* is the name of the ASCII file you created that contains your list of documents):

Note: Test to make sure that your documents were queued successfully. Type:

```
/usr/IMNSearch/bin/itestaix -s server -x indexname
```

The number after **Number of indexing requests scheduled** should equal the number of documents in your index.

7. **Start the Index Updating Process to Build your Index**

   Start the index updating process. This will take the documents that are in your document update list, index them, and put the results into the empty index to build your final complete index.

   Note: Indexing may take a significant amount of time to complete. You **CANNOT** move onto the Update the Registration Table step until indexing is complete. Use the status command below to tell when indexing is done.

   Use the following update command:

   ```
   /usr/IMNSearch/bin/iteupdix -s server -x index_name
   ```

   Note: Test to make sure that your documents were indexed successfully. Type:

   ```
   /usr/IMNSearch/bin/itestaix -s server -x indexname
   ```

   The number after **Number of documents in the primary index** should equal the number of documents in your index.

8. **Update the Registration Table**

   Next you need to register the new index in the registration table of your development computer so that the search service knows the index exists and you can do a test searches of the index.

   To update the registration table on the development computer, do the following (all on one command line):

   Note: There must be a final slash (**/**) after the *application_name*.

   ```
   /usr/IMNSearch/bin/itedomap
       -p /var/docsearch/indexes -c -x index_name
       -sp /doc_link/locale/application_name/
       -ti index_title
   ```

   The *locale* variable is the name of the language directory under **/usr/share/man/info** where the index's documents are stored. The variable *index_name* is the name of your index, and *index_title* is the title of your index. The title is the text you want the user to see in the bottom of the search form when they are selecting which indexes to search. Remember that the title should be written using the same language and codeset as the documents inside the index.

   Additionally, for index titles, it is recommended that you specify the title as an HTML link. The title will then appear as a link in the search form. This allows a user to click on the title in the search form to open the first document in the index for reading.

Note:

Every web server has an internal document home directory where it starts its search for documents. When the Documentation Library Service is installed and configured, a filesystem link is placed in this directory. This link points to the standard location of documents in the AIX filesystem: **/usr/share/man/info**. Since your web server will automatically go to this location to find your documents, the search engine only needs the portion of the document path from this location forward.

The link that the Documentation Library Service puts into your web server's starting directory is:

doc_link -> /usr/share/man/info

Your web server will be able to serve the documentation with URLs like:

http://your.machine.name/doc_link/en_US/calculator/user/doc1.html

For example, you might want the title of your index to be `Calculator Application Manuals`. You have three documents(manuals) inside this one index which is named `cal413en`. You decide that when the title link is clicked it would be best for the Beginners Guide document to be opened. So, you would insert in the title the URL that opens the Beginners Guide document. If you would normally type the URL `/doc_link/en_US/calculator/user/doc1.htm` (doc_link is the link to the **/usr/share/man/info** directory) to open the Beginners Guide document, you would use the following update command (all on one command line):

```
/usr/IMNSearch/bin/itedomap
    -p /var/docsearch/indexes -c -x cal413en
    -sp /doc_link/en_US/calculator/
    -ti <A HREF='/doc_link/en_US/calculator/user/doc1.htm'>Calculator Application Manuals</A>
```

9. **Copy your HTML Documents from the Build Directory into the Documentation Directory**

   You must now copy your HTML documents into the location where they can be read by your users. Your documents should be placed under the directory **/usr/share/man/info/***locale*/*application_name*/*index_name*. Using our example, the Calculator Application's English documents would be placed in `/usr/share/man/info/en_US/calculator/`.

   a. Find out if the language directory **/usr/share/man/info/** *locale* already exists. If it does not exist, create it. When you create this directory, make sure that it is executable and readable by all users.

      Using our example, the English directory is named: `/usr/share/man/info/en_US`.

   b. Create your application directory under the language directory. The directory structure should now look like: **/usr/share/man/info/***locale*/*application_name*.

      Using our example, the application's directory is named: `/usr/share/man/info/en_US/calculator`.

   c. Copy your documents and place them under the application directory you just created. The directory structure should now look like:

      `/usr/share/man/info/`*locale*/*application_name*/*documents*.

      Using our example, you would use the following command to copy the calculator's documents from the build directory into the directory where they will be read by users.

      ```
      cp -R /usr/work/calculator/* /usr/share/man/info/en_US/calculator
      ```

      The calculator's documents would then end up in these locations:

      ```
      /usr/share/man/info/en_US/calculator/user/doc1.html
      /usr/share/man/info/en_US/calculator/user/doc2.html
      /usr/share/man/info/en_US/calculator/admin/doc3.html
      ```

10. **Test your Index**

    You have now completed the creation and registration of an index on this development computer. You should now test the index by opening the search form, selecting the new index for search, and searching for words that you know are in the index. If the index does not work properly and you need to remove it so you can build it again, go to the section called Removing Indexes in your

Documentation (“Removing Indexes of your Documentation” on page 624). When you are satisfied that the index is working correctly, go on to the next step

11. Final Step
    - If this development computer where you created the index is also your real documentation server computer, you are now done with creating an index.
    - If you are an application developer and you were creating this index for inclusion in your application’s installp install package, skip to the section titled “Packaging your Application’s Documentation” on page 624.
    - If this development computer is not your documentation server, you now need to copy the new index to your documentation server and register it there. To do this, complete the following steps on the computer where you just created your index:
      a. Type the command:
         ```
         cd /usr/docsearch/indexes
         ```
      b. An index is not a single file, it is really a collection of files. You need to create a tar file that contains copies all of the files that make up your index. To create the tar file, type this command:
         ```
         tar cvf index_name.tar index_name
         ```
      c. Next move this tar file to the documentation server by using the ftp command to put it into the **/usr/docsearch/indexes** directory on the destination machine.

         Note: Be sure to transfer the tar file in binary mode.
      d. Log on to the destination documentation server as root.
      e. Type the command:
         ```
         cd /usr/docsearch/indexes
         ```
      f. Untar the tar file.
         ```
         tar xvf index_name.tar
         ```
      g. Change the ownership of the indexes.
         ```
         chown -R imnadm:imnadm index_name
         ```
      h. Stop the search server.
         ```
         /usr/IMNSearch/bin/itess -stop search
         ```
      i. Update the master table.

         Type the following command, all on one command line:
         ```
         /usr/IMNSearch/bin/itemtupd -m /etc/IMNSearch
             -i /usr/docsearch/indexes/index_name/data
             -w /usr/docsearch/indexes/index_name/work
             -n index_name
         ```
      j. Restart the search server.
         ```
         /usr/IMNSearch/bin/itess -start search
         ```
      k. Next, you need to register the new index in the registration table of your documentation server computer so that the search service knows the index exists and you can do a test search of the index.

         To update the registration table on the development computer, do the following (all on one command line):

         Note: The following command must end with a slash (**/**) after the *application_name*.
         ```
         /usr/IMNSearch/bin/itedomap
             -p /var/docsearch/indexes -c -x index_name
             -sp /doc_link/locale/application_name/
             -ti index_title
         ```

         The *locale* variable is the name of the language directory under **/usr/share/man/info** where the index’s documents are stored. The variable *index_name* is the name of your index, and *index_title* is the search from title of your index. The title is the text you want the user to see in

the bottom of the search form when they are selecting which indexes to search. Remember that the title should be written using the same language and codeset as the documents inside the index.

Additionally, for index titles, it is recommended that you specify the title as an HTML link. The title will then appear as a link in the search form. This allows a user to click on the title in the search form to open your primary document in the index for reading.

For example, you might want the title of your index to be `Calculator Application Manuals`. You have three documents (manuals) inside this one index which is named `cal413en`. You decide that when the title link is clicked it would be best for the Beginners Guide document to be opened. So, you would insert in the title the URL that opens the Beginners Guide document. If you would normally type the URL `/doc_link/en_US/calculator/user/doc1.htm` (doc_link is the link to the **/usr/share/man/info** directory) to open the Beginners Guide document, you would use the following update command (all on one command line):

```
/usr/IMNSearch/bin/itedomap
    -p /var/docsearch/indexes -c -x cal413en
    -sp /doc_link/en_US/calculator/
    -ti <A HREF='/doc_link/en_US/calculator/user/doc1.htm'>Calculator Application Manuals</A>
```

- You have now finished the copy and registration of the index on the documentation server. You should do test searches of the index to make sure it is working correctly.

Removing Indexes of your Documentation

You cannot just delete files to remove an index from a server. This will leave the search service corrupted. Use the following steps to remove an index (replacing *index_name* with the name of the index you wish to remove):

1. Delete the index.

   ```
   /usr/IMNSearch/bin/itedelix -s server -x index_name
   ```

2. Remove the index entry in the registration table.

   ```
   /usr/IMNSearch/bin/itedomap -p /var/docsearch/indexes -d -x index_name
   ```

3. Delete the empty index directories that held the index files:

   ```
   rm -r /usr/docsearch/indexes/index_name
   ```

Packaging your Application's Documentation

1. "Include a Search Index"
2. "Register your Documentation" on page 626
3. "Create an install package" on page 626

Include a Search Index

To include a search index in your application's **installp** installation package, you will need to complete the following steps:

Note: You must repeat these steps for each separately installable fileset in your package that contains one or more indexes.

1. **Create the install script**

   You must perform the following steps to create a registration script. This script will automatically register your indexes with the Documentation Library Service during the installation of your application's **installp** installation package. You will be using and modifying an example script to create your own registration script.

   a. Make a copy of the example script **/usr/docsearch/tools/index_config.sh.** You can use any name for the copy.

b. Edit the script and change:

Note: The script is designed to install one or more indexes. In each of the following variables, replace the ***X*** character with the number for the index you are specifying.

1) **index_type** to DBCS if you are registering double-byte codeset indexes.
2) **indexdir_name_*X*** to the name of your index (repeat for each index).
3) **index_title_*X*** to the title of your index.
4) **index_loc_*X*** to /usr/docsearch/indexes. This is where **installp** will be placing your index when your application is installed.
5) **document_loc_*X*** to the temporary portion of the document path. This path segment must begin **and** end with a slash (**/**).

Example:
To install the indexes Book1Sen and Book2Sen**,** which are being installed in /usr/docsearch/indexes/Book1Sen and /usr/docsearch/indexes/Book2Sen, have the titles Book #1 and Book #2, and whose documents are in /usr/share/man/info/en_US/calculator/**...** you might have lines in the script like:

```
indexdir_name_1="Book1Sen"
indexdir_name_2="Book2Sen"

index_title_1="<A HREF="/doc_link/en_US/calculator/Book1S.html">Book #1</A>"
index_title_2="<A HREF="/doc_link/en_US/calculator/Book2S.html">Book #2</A>"

index_loc_1="/usr/docsearch/indexes/Book1Sen"
index_loc_2="/usr/docsearch/indexes/Book2Sen"

document_loc_1="/doc_link/en_US/"
document_loc_2="/doc_link/en_US/"
```

c. Delete all other index*XXX* variable assignments from the script. There should only be as many lines of the form **indexdir_name_X="..."** as there are indexes you want to install. The same holds true for **index_title_X**, **index_loc_X**, and **document_loc_X**.

2. **Create the uninstall script**

Create the **uninstall** script that will cleanly unregister your index if your application is uninstalled.

a. Make a copy of the unconfig script in **/usr/docsearch/tools/index_unconfig.sh**
b. Edit the script and change *index_type* to DBCS if the indexes you are unregistering are double-byte indexes.
c. Edit the script and change **indexdir_name_*X*** to the name of your index (repeat for each index).
d. Delete all other **indexdir_name_*X*** variable assignments from the script. There should only be as many lines of the form **indexdir_name_X="..."** as there are indexes you want to uninstall.

3. **Create the pre_rm script**

Create the **pre_rm script** that will cleanly unregister your index when your application is reinstalled using a force install or updated in preparation for installing new versions of your index.

a. Make a copy of the **pre_rm script** that is in **/usr/docsearch/tools/index_pre_rm.sh**
b. Edit the script and change *index_type* to DBCS if you are unregistering any double-byte indexes.
c. Edit your copy of the script and change **indexdir_name_X** to the name of your index (repeat for each index).

Example: If you have two indexes with the names cal413en and cal567en, your copy of the **pre_rm script** would have lines like:

```
indexdir_name_1="cal413en"
indexdir_name_2="cal567en"
```

d. Delete all other **indexdir_name_X** variable assignments from the script. There should only be as many lines of the form **indexdir_name_X="..."** as there are indexes in your fileset.

Register your Documentation

To have your application's **installp** installation package automatically register your documentation into a view you will need to complete the following steps:

1. **Ship your configuration file to the appropriate directory in /usr/docsearch/views**

   See the section titled Create a View Set Configuration File.
2. **Create a view definition file for every view in which you want your documents to appear**

   See the section titled Create a View Definition File.
3. **Modify the install script** After the call to **/usr/sbin/index_config.sh**, put a line to register a view definition file for each view into which you want to register your documentation.

   See the section titled Register the Contents of each of your View Definition Files.
4. **Modify the uninstall and pre_rm scripts** After the call to **/usr/sbin/index_config.sh**, put a line to unregister a view definition file for each view into which you registered your documentation.

   See the section titled Register the Contents of each of your View Definition Files.

Create an install package

Create a normal install package for your documentation or application. If you need instructions on how to create an install package, see "Chapter 20. Packaging Software for Installation" on page 567.

In addition to the normal packaging steps, do the following:

1. Place the **install** script in your **installp** package so that it will be run in your post-install process when the fileset containing the index is installed.
2. Place the **uninstall** script in your **installp** package so that it will be run in your uninstall process when the fileset containing the index is uninstalled.
3. Place the **pre_rm** script in your **installp** package so that it will be run when the fileset containing the index is uninstalled.
4. If you are using configuration files, have your package create your application's **config** directory, put your configuration file(s) there, and set permissions for the directories and configuration files.
5. During installation, have your package install your documentation and indexes.

Packaging Book Guidelines

By using the Printfile tag in the VDF, you have the option of defining a single printable file which contains all of the files that make up your book. This file will then appear within the Print Tool page of the library service so that users can download this file for printing on their local printer. For further information on using the Printfile tag and the other packaging tasks, see "Making your Documents Printable" on page 609.

Chapter 22. Software Vital Product Data (SWVPD)

Information about a software product and its installable options is maintained in the Software Vital Product Data (SWVPD) database. The SWVPD consists of a set of commands and the Object Data Manager (ODM) object classes for the maintenance of software product information. The SWVPD commands are provided for the user to query (**lslpp**) and verify (**lppchk**) installed software products. The ODM object classes define the scope and format of the software product information that is maintained.

The **installp** command uses the Object Data Manager to maintain the following information in the SWVPD database:

- The name of the software product (for example, AIXwindows).
- The version of the software product, which indicates the operating system upon which it operates.
- The release level of the software product, which indicates changes to the external programming interface of the software product.
- The modification level of the software product, which indicates changes that do not affect the software product's external interface.
- The fix level of the software product, which indicates small updates that are to be built into a regular modification level at a later time.
- The fix identification field.
- The names, checksums, and sizes of the files that make up the software product or option.
- The state of the software product: available, applying, applied, committing, committed, rejecting, or broken.

Object Classes

The information in the **lpp**, **inventory**, **history**, and **product** object classes comprises the SWVPD for an installed software product. These object classes are stored in the following directories:

/etc/objrepos	/ (root) part of the installable software product
/usr/lib/objrepos	/usr part of the installable software product
/usr/share/lib/objrepos	/usr/share part of the installable software product

Any of the ODM commands and subroutines can be used with these object classes. All of the object classes and defined values for the SWVPD are in the **swvpd.h** header file. A constant that defines an object class attribute is valid for only that object class.

lpp Object Class (LPP_TABLE)	The **lpp** object class contains information about the installed software products, including the current software product state.
inventory Object Class (INVENTORY_TABLE)	The **inventory** object class contains information about the files associated with a software product.
history Object Class (HIST_TABLE)	The **history** object class contains historical information about the installation and updates of software products.
product Object Class (PRODUCT_TABLE)	The **product** object class contains product information about the installation and updates of software products and their prerequisites.

Files

/etc/objrepos	Contains the four object classes used by the SWVPD for the **/** (root) part of the installable software product.
/usr/lib/objrepos	Contains the four object classes used by the SWVPD for the **/usr** part of the installable software product.
/usr/share/lib/objrepos	Contains the four object classes used by the SWVPD for the **/usr/share** part of the installable software product.

Chapter 23. Source Code Control System (SCCS)

The Source Code Control System (SCCS) is a complete system of commands that allows specified users to control and track changes made to an SCCS file. SCCS files allow several versions of the same file to exist simultaneously, which can be helpful when developing a project requiring many versions of large files. The SCCS commands support Multibyte Character Set (MBCS) characters.

Introduction to SCCS

The SCCS commands form a complete system for creating, editing, converting, or changing the controls on SCCS files. An SCCS file is any text file controlled with SCCS commands. All SCCS files have the prefix **s.**, which sets them apart from regular text files.

Attention: Using non-SCCS commands to edit SCCS files can damage the SCCS files.

Use SCCS commands on an SCCS file. If you wish to look at the structure of an SCCS file, use the **pg** command or a similar command to view its contents. However, do not use an editor to directly change the file.

To change text in an SCCS file, use an SCCS command (such as the **get** command) to obtain a version of the file for editing, and then use any editor to modify the text. After changing the file, use the **delta** command to save the changes. To store the separate versions of a file, and control access to its contents, SCCS files have a unique structure.

An SCCS file is made up of three parts:

- Delta table
- Access and tracking flags
- Body of the text

Delta Table in SCCS files

Instead of creating a separate file for each version of a file, the SCCS file system only stores the changes for each version of a file. These changes are referred to as *deltas*. The changes are tracked by the delta table in every SCCS file.

Each entry in the delta table contains information about who created the delta, when they created it, and why they created it. Each delta has a specific SID (SCCS IDentification number) of up to four digits. The first digit is the release, the second digit the level, the third digit the branch, and the fourth digit the sequence.

An example of an SID number is:

```
SID = 1.2.1.4
```

that is, release 1, level 2, branch 1, sequence 4.

No SID digit can be 0, so there cannot be an SID of 2.0 or 2.1.2.0, for example.

Each time a new delta is created, it is given the next higher SID number by default. That version of the file is built using all the previous deltas. Typically, an SCCS file grows sequentially, so each delta is identified only by its release and level. However, a file may branch and create a new subset of deltas. The file then has a trunk, with deltas identified by release and level, and one or more branches, which have deltas containing all four parts of an SID. On a branch, the release and level numbers are fixed, and new deltas are identified by changing sequence numbers.

Note: A file version built from a branch does not use any deltas placed on the trunk after the point of separation.

Control and Tracking Flags in SCCS Files

After the delta table in an SCCS file, a list of flags starting with the @ (at sign) define the various access and tracking options of the SCCS file. Some of the SCCS flag functions include:

- Designating users who may edit the files
- Locking certain releases of a file from editing
- Allowing joint editing of the file
- Cross-referencing changes to a file

Body of an SCCS file

The SCCS file body contains the text for all the different versions of the file. Consequently, the body of the file does not look like a standard text file. Control characters bracket each portion of the text and specify which delta created or deleted it. When the SCCS system builds a specific version of a file, the control characters indicate the portions of text that correspond to each delta. The selected pieces of text are then used to build that specific version.

SCCS Flag and Parameter Conventions

In most cases, SCCS commands accept two types of parameters:

flags	Flags consist of a - (minus sign), followed by a lowercase character, which is sometimes followed by a value. Flags control how the command operates.
File or *Directory*	These parameters specify the file or files with which the command operates. Using a directory name as an argument specifies all SCCS files in that directory.

File or directory names cannot begin with a - (minus sign). If you specify this sign by itself, the command reads standard input or keyboard input until it reaches an end-of-file character. This is useful when using pipes that allow processes to communicate.

Any flags specified for a command apply to all files on the command line and are processed before any other parameters to that command. Flag placement in the command line is not important. Other parameters are processed left to right. Some SCCS files contain flags that determine how certain commands operate on the file. See the **admin** command description of SCCS header flags for more information.

Creating, Editing, and Updating an SCCS File

You can create, edit, and update an SCCS file using the **admin**, **get**, and **delta** commands.

Creating an SCCS File

admin Creates an SCCS file or changes an existing SCCS file.

- To create an empty SCCS file named `s.test.c`, enter:

  ```
  admin -n s.test.c
  ```

  Using the **admin** command with the **-n** flag creates an empty SCCS file.
- To convert an existing text file into an SCCS file, enter:

```
admin -itest.c s.test.c
There are no SCCS identification keywords in the file (cm7)

ls
s.test.c test.c
```

If you use the **-i** flag, the **admin** command creates delta 1.1 from the specified file. Once delta 1.1 is created, rename the original text file so it does not interfere with SCCS commands (it will act as a backup):

```
mv test.c back.c
```

The message `There are no SCCS identification keywords in the file (cm7)` does not indicate an error.

- To start the `test.c` file with a release number of 3.1, use the **-r** flag with the **admin** command, as follows:

```
admin -itest.c -r3 s.test.c
```

Editing an SCCS file

Attention: Do not edit SCCS files directly with non-SCCS commands, or you can damage the SCCS files.

get Gets a specified version of an SCCS file for editing or compiling.

1. To edit an SCCS file, enter the **get** command with the **-e** flag to produce an editable version of the file:

```
get -e s.test.c
1.3
new delta 1.4
67 lines

ls
p.test.c s.test.c test.c
```

The **get** command produces two new files, `p.test.c` and `test.c`. The editable file is `test.c`. The `p.test.c` file is a temporary, uneditable file used by SCCS to keep track of file versions. It will disappear when you update your changes to the SCCS file. Notice also that the **get** command prints the SID of the version built for editing, the SID assigned to the new delta when you update your changes, and the number of lines in the file.

2. Use any editor to edit `test.c`, for example:

```
ed test.c
```

You can now work on your actual file. Edit this file as often as you wish. Your changes will not affect the SCCS file until you choose to update it.

3. To edit a specific version of an SCCS file with multiple versions, enter the **get** command with the **-r** flag :

```
get -r1.3 s.test.c
1.3
67 lines

get -r1.3.1.4 s.test.c
1.3.1.4
50 lines
```

Updating an SCCS File

delta Adds a set of changes (deltas) to the text of an SCCS file.

1. To update the SCCS file and create a new delta with the changes you made while editing, use the **delta** command:

   ```
   $delta s.test.c
   Type comments, terminated with EOF or a blank line:
   ```

2. The **delta** command prompts you for comments to be associated with the changes you made. For example, enter your comments, and then press the Enter key twice:

   ```
   No id keywords (cm7)
   1.2
   5 lines inserted
   6 lines deleted
   12 lines unchanged
   ```

   The **delta** command updates the s.prog.c file with the changes you made to the `test.c` file. The **delta** command tells you that the SID of the new version is 1.2, and that the edited file inserted 5 lines, deleted 6 lines, and left 12 lines unchanged from the previous version.

Controlling and Tracking SCCS File Changes

The SCCS command and file system are primarily used to control access to a file and to track who altered a file, why it was altered, and what was altered.

Controlling Access to SCCS files

Three kinds of access can be controlled in an SCCS file system:

- File access
- User access (“User Access Controls”)
- Version access (“Version Access Controls”).

File Access Controls

Directories containing SCCS files should be created with permission code 755 (read, write, and execute permissions for owner; read and execute permissions for group members and others). The SCCS files themselves should be created as read-only files (444). With these permissions, only the owner can use non-SCCS commands to modify SCCS files. If a group can access and modify the SCCS files, the directories should have group write permission.

User Access Controls

The **admin** command with the **-a** flag can designate a group of users that can make changes to the SCCS file. A group name or number can also be specified with this flag.

Version Access Controls

The **admin** command can lock, or prevent, various versions of a file from being accessed by the **get** command by using header flags.

-fc Sets a ceiling on the highest release number that can be retrieved
-ff Sets a floor on the lowest release number that can be retrieved
-fl Locks a particular release against being retrieved

Tracking Changes to an SCCS File

There are three ways to track changes to an SCCS file:

- Comments associated with each delta
- Modification Request (MR) numbers
- The SCCS commands.

Tracking Changes with Delta Comments

After an SCCS file is updated and a new delta created, the system prompts for comments to be associated with that delta. These comments can be up to 512 characters long and can be modified with the **cdc** command.

cdc Changes the comments associated with a delta

The **get** command with the **-l** flag prints out the delta table and all the delta comments for any version of a file. In addition to storing the comments associated with a delta, the delta table automatically stores the time and date of the last modification, the real user ID at the time of the modification, the serial numbers of the delta and its predecessor, and any MR numbers associated with the delta.

Tracking Changes with Modification Request Numbers

The **admin** command with the **-fv** flag prompts for MR numbers each time a delta is created. A program can be specified with the **-fv** flag to check the validity of the MR numbers when an attempt is made to create a new delta in the SCCS file. If the MR validity-checking program returns a nonzero exit value, the update will be unsuccessful.

The MR validity-checking program is created by the user. It can be written to track changes made to the SCCS file and index them against any other database or tracking system.

Tracking Changes with SCCS commands

sccsdiff Compares two SCCS files and prints their differences to standard output

The **delta** command with the **-p** flag acts the same as the **sccsdiff** command when the file is updated. Both of these commands allow you to see what changes have been made between versions.

prs Formats and prints specified portions of an SCCS file to standard output

This command allows you to find the differences in two versions of a file.

Detecting and Repairing Damaged SCCS Files

You can detect and repair damaged SCCS files using the **admin** command.

Procedure

1. Check SCCS files on a regular basis for possible damage. Any time an SCCS file is changed without properly using SCCS commands, damage may result to the file. The SCCS file system detects this damage by calculating the checksum and comparing it with the one stored in the delta table. Check for damage by running the **admin** command with the **-h** flag on all SCCS files or SCCS directories as shown:

   ```
   admin -h s.file1 s.file2 ...
   ```

   OR

   ```
   admin -h directory1 directory2 ...
   ```

   If the **admin** command finds a file where the computed checksum is not equal to the checksum listed in the SCCS file header, it displays this message:

   ```
   ERROR [s.filename]:
   1255-057 The file is damaged. (co6)
   ```

2. If a file was damaged, try to edit the file again or read a backup copy. Once the checksum has been recalculated, any remaining damage will be undetectable by the **admin** command.

Note: Using the **admin** command with the **-z** flag on a damaged file can prevent future detection of the damage.

3. After fixing the file, run the **admin** command with the **-z** flag and the repaired file name:

```
admin -z s.file1
```

List of Additional SCCS Commands

Attention: Using non-SCCS commands with SCCS files can damage the SCCS files.

The following SCCS commands complete the system for handling SCCS files:

rmdel	Removes the most recent delta on a branch from an SCCS file.
sact	Displays current SCCS file editing status.
sccs	Administration program for the SCCS system. The **sccs** command contains a set of pseudo-commands that perform most SCCS services.
sccshelp	Explains an SCCS error message or command.
unget	Cancels the effect of a previous use of the **get -e** command.
val	Checks an SCCS file to see if its computed checksum matches the checksum listed in the header.
vc	Substitutes assigned values in place of identification keywords.
what	Searches a system file for a pattern and displays text that follows it.

Chapter 24. Subroutines, Example Programs, and Libraries

This chapter provides information about what subroutines are, how to use them, and where they are stored.

Subroutines are stored in libraries to conserve storage space and to make the program linkage process more efficient. A *library* is a data file that contains copies of a number of individual files and control information that allows them to be accessed individually. The libraries are located in the **/usr/ccs/lib** and **/usr/lib** directories. By convention, most of them have names of the form **lib***name***.a** where *name* identifies the specific library.

All include statements should be near the beginning of the first file being compiled, usually in the declarations section before **main( )**, and must occur before using any library functions. For example, use the following statement to include the **stdio.h** file:

```
#include <stdio.h>
```

You do not need to do anything special to use subroutines from the Standard C library (**libc.a**). The **cc** command automatically searches this library for subroutines that a program needs. However, if you use subroutines from another library, you must tell the compiler to search that library. If your program uses subroutines from the library **lib***name***.a**, compile your program with the flag **-l***name* (lowercase L). The following example compiles the program `myprog.c`, which uses subroutines from the **libdbm.a** library:

```
cc myprog.c -ldbm
```

You can specify more than one **-l** (lowercase L) flag. Each flag is processed in the order specified.

If you are using a subroutine that is stored in the Berkeley Compatibility Library, bind to the **libbsd.a** library *before* binding to the **libc.a** library, as shown in the following example:

```
cc myprog.c -lbsd
```

When an error occurs, many subroutines return a value of -1 and set an external variable named **errno** to identify the error. The **sys/errno.h** file declares the **errno** variable and defines a constant for each of the possible error conditions.

In this documentation, all system calls are described as *subroutines* and are resolved from the **libc.a** library. The programming interface to system calls is identical to that of subroutines. As far as a C Language program is concerned, a system call is merely a subroutine call. The real difference between a system call and a subroutine is the type of operation it performs. When a program invokes a system call, a protection domain switch takes place so that the called routine has access to the operating system kernel's privileged information. The routine then operates in kernel mode to perform a task on behalf of the program. In this way, access to the privileged system information is restricted to a predefined set of routines whose actions can be controlled.

Notes:

1. The following list represents the wString routines that are obsolete for the 64 bit **libc.a**. Their corresponding 64 bit **libc.a** equivalents are included. The routines for the 32 bit libc.a can be found in the wstring Subroutine. The corresponding routines for the 64 bit **libc.a** can be found in the List of Wide Character Subroutines ("List of Wide Character Subroutines" on page 503).

32 Bit only	64 Bit Equivalent
wstrcat	wcscat
wstrchr	wcschr
wstrcmp	wcscoll
wstrcpy	wcscpy
wstrcspn	wcscspn
wstrdup	Not available and has no

```
                          equivalents in the 64 bit libc.a
wstrlen                   wcslen
wstrncat                  wcsncat
wstrncpy                  wcsncpy
wstrpbrk                  wcspbrk
wstrrchr                  wcsrchr
wstrspn                   wcsspn
wstrtok                   wcstok
```

2. All programs that handle multibyte characters, wide characters, or locale-specific information must call the **setlocale** subroutine at the beginning of the program. See “National Language Support Subroutines Overview” on page 339 for more information.
3. Programming in a multi-threaded environment requires reentrant subroutines to ensure data integrity. See the ″List of Multi-threaded Programming Subroutines″ (“List of Multi-threaded Programming Subroutines” on page 646)

128-Bit Long Double Floating-Point Data Type

The AIX operating system supports a 128-bit long double data type that provides greater precision than the default 64-bit long double data type. The 128-bit data type can handle up to 31 significant digits (compared to 17 handled by the 64-bit long double). However, while this data type can store numbers with more precision than the 64-bit data type, it does not store numbers of greater magnitude.

The following special issues apply to the use of the 128-bit long double data type:

- Compiling programs that use the 128-bit long double data type
- Compliance with the IEEE 754 standard
- Implementing the 128-bit long double format
- Values of numeric macros

Compiling Programs that Use the 128-bit Long Double Data Type

To compile C programs that use the 128-bit long double data type, use the **xlc128** command. This command is an alias to the **xlc** command with support for the 128-bit data type. The **xlc** command supports only the 64-bit long double data type.

The standard C library, **libc.a** provides replacements for **libc.a** routines which are implicitly sensitive to the size of long double. Link with the **libc.a** library when compiling applications that use the 64-bit long double data type. Link applications that use 128-bit long double values with both the **libc128.a** and **libc.a** libraries. When linking, be sure to specify the **libc128.a** library before the **libc.a** library in the library search order.

Compliance with IEEE 754 Standard

The 64-bit implementation of the long double data type is fully compliant with the IEEE 754 standard, but the 128-bit implementation is not. Use the 64-bit implementation in applications that must conform to the IEEE 754 standard.

The 128-bit implementation differs from the IEEE standard for long double in the following ways:

- Supports only round-to-nearest mode. If the application changes the rounding mode, results are undefined.
- Does not fully support the IEEE special numbers NaN and INF.
- Does not support IEEE status flags for overflow, underflow, and other conditions. These flags have no meaning for the 128-bit long double inplementation.

Implementing the 128-Bit Long Double Format

A 128-bit long double number consists of an ordered pair of 64-bit double-precision numbers. The first member of the ordered pair contains the high-order part of the number, and the second member contains the low-order part. The value of the long double quantity is the sum of the two 64-bit numbers.

Each of the two 64-bit numbers is itself a double-precision floating-point number with a sign, exponent, and significand. Typically the low-order member has a magnitude that is less than 0.5 units in the last place of the high part, so the values of the two 64-bit numbers do not overlap and the entire significand of the low-order number adds precision beyond the high-order number.

This representation results in several issues that must be considered in the use of these numbers:

- The exponent range is the same as that of double precision. Although the precision is greater, the magnitude of representable numbers is the same as 64-bit double precision.
- As the absolute value of the magnitude decreases (near the denormal range), the additional precision available in the low-order part also decreases. When the value to be represented is in the denormal range, this representation provides no more precision than the 64-bit double-precision data type.
- The actual number of bits of precision can vary. If the low-order part is much less than 1 ULP of the high-order part, significant bits (either all 0's or all 1's) are implied between the significands of the high-order and low-order numbers. Certain algorithms that rely on having a fixed number of bits in the significand can fail when using 128-bit long double numbers.

Values of Numeric Macros

Because of the storage method for the long double data type, more than one number can satisfy certain values that are available as macros.The representation of 128-bit long double numbers means that the following macros required by standard C in the **values.h** file do not have clear meaning:

- Number of bits in the mantissa (**LDBL_MANT_DIG**)
- Epsilon (**LBDL_EPSILON**)
- Maximum representable finite value (**LDBL_MAX**)

Number of Bits in the Mantissa

The number of bits in the significand is not fixed, but for a correctly formatted number (except in the denormal range) the minimum number available is 106. Therefore, the value of the **LDBL_MANT_DIG** macro is 106.

Epsilon

The ANSI C standard defines the value of epsilon as the difference between 1.0 and the least representable value greater than 1.0, that is, *b***(1-*p*), where *b* is the radix (2) and *p* is the number of base *b* digits in the number. This definition requires that the number of base *b* digits is fixed, which is not true for 128-bit long double numbers.

The smallest representable value greater than 1.0 is this number:

```
0x3FF0000000000000, 0x0000000000000001
```

The difference between this value and 1.0 is this number:

```
0x0000000000000001, 0x0000000000000000
0.4940656458412465441765687928682213E-323
```

Because 128-bit numbers usually provide at least 106 bits of precision, an appropriate minimum value for *p* is 106. Thus, *b***(1-*p*) and 2**(-105) yield this value:

```
0x3960000000000000, 0x0000000000000000
0.2465190328815661891911651766508707 0E-31
```

Both values satisfy the definition of epsilon according to standard C. The long double subroutines use the second value because it better characterizes the accuracy provided by the 128-bit implementation.

Maximum Long Double Value

The value of the **LDBL_MAX** macro is the largest 128-bit long double number that can be multiplied by 1.0 and yield the original number. This value is also the largest finite value that can be generated by primitive operations, such as multiplication and division:

```
0x7FEFFFFFFFFFFFFF, 0x7C8FFFFFFFFFFFFF
0.17976931348623158079372897140530 23E+309
```

List of Character Manipulation Subroutines

The character manipulation functions and macros test and translate ASCII characters.

These functions and macros are of three kinds:

- Character testing
- Character translation
- Miscellaneous character manipulation

The "Programming Example for Manipulating Characters" on page 649 illustrates some of the character manipulation routines.

Character Testing

Use the following functions and macros to determine character type. Punctuation, alphabetic, and case-querying functions values depend on the current collation table.

The **ctype** subroutines contain the following functions:

isalpha	Is character alphabetic?
isalnum	Is character alphanumeric?
isupper	Is character uppercase?
islower	Is character lowercase?
isdigit	Is character a digit?
isxdigit	Is character a hex digit?
isspace	Is character a blank-space character?
ispunct	Is character a punctuation character?
isprint	Is character a printing character, including space?
isgraph	Is character a printing character, excluding space?
iscntrl	Is character a control character?
isascii	Is character an integer ASCII character?

Character Translation

The **conv** subroutines contain the following functions:

toupper	Converts a lowercase letter to uppercase
_toupper	(Macro) Converts a lowercase letter to uppercase
tolower	Converts an uppercase letter to lowercase
_tolower	(Macro) Converts an uppercase letter to lowercase
toascii	Converts an integer to an ASCII character

Miscellaneous Character Manipulation

getc, **fgetc**,**getchar**, **getw**	Get a character or word from an input stream
putc,**putchar**, **fputc**, **putw**	Write a character or word to a stream

List of Executable Program Creation Subroutines

The list of executable program creation services consists of subroutines that support a group of commands. These commands and subroutines allow you to create, compile, and work with files in order to make your programs run.

_end, **_text, _edata**	Define the last location of a program
confstr	Determines the current value of a specified system variable defined as a string
getopt	Gets flag letters from the argument vector
ldopen, **ldaopen**	Open a common object file
ldclose, **ldaclose**	Close a common object file
ldahread	Reads the archive header of a member of an archive file
ldfhread	Reads the file header of a common object file
ldlread, **ldlinit**, **ldlitem**	Read and manipulate line number entries of a common object file function
ldshread, **ldnshread**	Read a section header of a common object file
ldtbread	Reads a symbol table entry of a common object file
ldgetname	Retrieves a symbol name from a symbol table entry or from the string table
ldlseek, **ldnseek**	Seek to line number entries of a section of a common object file
ldohseek	Seek to the optional file header of a common object file
ldrseek, **ldnrseek**	Seek to the relocation information for a section of a common object file
ldsseek, **ldnsseek**	Seek to a section of a common object file
ldtbseek	Seeks to the symbol table of a common object file
ldtbindex	Returns the index of a particular common object file symbol table entry
load	Loads and binds an object module into the current process
unload	Unloads an object file
loadbind	Provides specific runtime resolution of a module's deferred symbols
loadquery	Returns error information from the **load** subroutine or the **exec** subroutine. Also provides a list of object files loaded for the current process
monitor	Starts and stops execution profiling
nlist	Gets entries from a name list
regcmp, **regex**	Compile and matche regular-expression patterns
setjmp, **longjmp**	Store a location
sgetl, **sputl**	Accesses long numeric data in a machine-independent fashion
sysconf	Determines the current value of a specified system limit or option

List of Files and Directories Subroutines

The system provides services to create files, move data into and out of files, and describe restrictions and structures of the file system. Many of these subroutines are the base for the system commands that have similar names. You can, however, use these subroutines to write new commands or utilities to help in the program development process, or to include in an application program.

The system provides subroutines for:

- Controlling Files
- "Working with Directories" on page 640
- "Manipulating File Systems" on page 641

Controlling Files

access, **accessx**, or **faccessx**	Determine accessibility of a file
fclear	Clears space in a file
fcntl, **dup**, or **dup2**	Control open file descriptors
fsync	Writes changes in a file to permanent storage
getenv	Returns the value of an environment variable
getutent, **getutid**, **getutline**, **pututline**, **setutent**, **endutent**, or **utmpname**	Access utmp file entries
getutid_r, **getutline_r**, **pututline_r**, **setutent_r**, **endutent_r**, or **utmpname_r**	Access utmp file entries
lseek or **llseek**	Move the read-write pointer in an open file
lockfx, **lockf**, or **flock**	Controls open file descriptor locks
mknod or **mkfifo**	Create regular, FIFO, or special files
mktemp or **mkstemp**	Construct a unique file name
open, **openx**, or **creat**	Return a file descriptor and creates files
pclose	Closes an open pipe
pipe	Creates an interprocess channel
popen	Initiates a pipe to a process
pathconf, **fpathconf**	Retrieve file implementation characteristics
putenv	Sets an environment variable
read, **readx**, **readv**, **readvx**	Read from a file or device
rename	Renames directory or file within a file system
statx, **stat**, **fstatx**, **fstat**, **fullstat**, **fullstat**	Get file status
tmpfile	Creates a temporary file
tmpnam or **tempnam**	Construct a name for a temporary file
truncate, **ftruncate**	Make a file shorter
umask	Gets and sets the value of the file creation mask
utimes or **utime**	Set file access or modification time
write, **writex**, **writev**, **writevx**	Write to a file or device

Working with Directories

chdir	Changes the current working directory
chroot	Changes the effective root directory
getwd, **getcwd**	Get the current directory path name
glob	Generates a list of path names to accessible files
globfree	Frees all memory associated with the *pglob* parameter
link	Creates additional directory entry for an existing file
mkdir	Creates a directory
opendir, **readdir**, **telldir**, **seekdir**, **rewinddir**, **closedir**	Performs operations on directories
readdir_r	Reads a directory
rmdir	Removes a directory
scandir, **alphasort**	Scan a directory
readlink	Reads the volume of a symbolic link
remove	Makes a file inaccessible by specified name
symlink	Creates a symbolic link to a file
unlink	Removes a directory entry

Manipulating File Systems

confstr	Determines the current value of a specified system variable defined by a string
fscntl	Manipulates file system control operations
getfsent, **getfsspec**, **getfsfile**, **getfstype**, **setfsent**, or **endfsent**	Get information about a file system
getfsent_r, **getfsspec_r**, **getfsfile_r**, **getfstype_r**, **setfsent_r**, or **endfsent_r**	Get information about a file system
getvfsent, **getvfsbytype**, **getvfsbyname**, **getvfsbyflag**, **setvfsent**, **endvfsent**	Get information about virtual file system entries
mnctl	Returns mount status information
quotactl	Manipulates disk quotas
statfs, **fstatfs**	Get the status of a file's file system
sysconf	Reports current value of system limits or options
sync	Updates all file systems information to disk
umask	Gets and sets the value of the file creation mask
vmount	Mounts a file system
umount, **uvmount**	Remove a virtual file system from the file tree

List of FORTRAN BLAS Level 1: Vector-Vector Subroutines

Level 1: vector-vector subroutines include:

SDOT, **DDOT**	Return the dot product of two vectors
CDOTC, **ZDOTC**	Return the complex dot product of two vectors, conjugating the first
CDOTU, **ZDOTU**	Return the complex dot product of two vectors
SAXPY, **DAXPY**, **CAXPY**, **ZAXPY**	Return a constant times a vector plus a vector
SROTG, **DROTG**, **CROTG**, **ZROTG**	Construct a Givens plane rotation
SROT, **DROT**, **CSROT**, **ZDROT**	Apply a plane rotation
SCOPY, **DCOPY**, **CCOPY**, **ZCOPY**	Copy vector *X* to *Y*
SSWAP, **DSWAP**, **CSWAP**, **ZSWAP**	Interchange vectors *X* and *Y*
SNRM2, **DNRM2**, **SCNRM2**, **DZNRM2**	Return the Euclidean norm of the *N*-vector stored in *X*() with storage increment *INCX*
SASUM, **DASUM**, **SCASUM**, **DZASUM**	Return the sum of absolute values of vector components
SSCAL, **DSCAL**, **CSSCAL**, **CSCAL**, **ZDSCAL**, **ZSCAL**	Scale a vector by a constant
ISAMAX, **IDAMAX**, **ICAMAX**, **IZAMAX**	Find the index of element having maximum absolute value
SDSDOT	Returns the dot product of two vectors plus a constant
SROTM, **DROTM**	Apply the modified Givens transformation
SROTMG, **DROTMG**	Construct a modified Givens transformation

List of FORTRAN BLAS Level 2: Matrix-Vector Subroutines

Level 2: matrix-vector subroutines include:

SGEMV, **DGEMV**,**CGEMV**, **ZGEMV**	Perform matrix-vector operation with general matrices
SGBMV, **DGBMV**, **CGBMV**, **ZGBMV**	Perform matrix-vector operations with general banded matrices
CHEMV, **ZHEMV**	Perform matrix-vector operations using Hermitian matrices
CHBMV, **ZHBMV**	Perform matrix-vector operations using a Hermitian band matrix
CHPMV,**ZHPMV**	Perform matrix-vector operations using a packed Hermitian matrix
SSYMV , **DSYMV**	Perform matrix-vector operations using a symmetric matrix
SSBMV , **DSBMV**	Perform matrix-vector operations using symmetric band matrix

SSPMV , **DSPMV**	Perform matrix-vector operations using a packed symmetric matrix
STRMV, **DTRMV**, **CTRMV**, **ZTRMV**	Perform matrix-vector operations using a triangular matrix
STBMV, **DTBMV**, **CTBMV**, **ZTBMV**	Perform matrix-vector operations using a triangular band matrix
STPMV, **DTPMV**, **CTPMV**, **ZTPMV**	Perform matrix-vector operations on a packed triangular matrix
STRSV, **DTRSV**, **CTRSV**, **ZTRSV**	Solve system of equations
STBSV, **DTBSV**, **CTBSV**, **ZTBSV**	Solve system of equations
STPSV, **DTPSV**, **CTPSV**, **ZTPSV**	Solve systems of equations
SGER, **DGER**	Perform rank 1 operation
CGERU, **ZGERU**	Perform rank 1 operation
CGERC,**ZGERC**	Perform rank 1 operation
CHER, **ZHER**	Perform Hermitian rank 1 operation
CHPR,**ZHPR**	Perform Hermitian rank 1 operation
CHPR2,**ZHPR2**	Perform Hermitian rank 2 operation
SSYR, **DSYR**	Perform symmetric rank 1 operation
SSPR, **DSPR**	Perform symmetric rank 1 operation
SSYR2 , **DSYR2**	Perform symmetric rank 2 operation
SSPR2 ,**DSPR2**	Perform symmetric rank 2 operation

List of FORTRAN BLAS Level 3: Matrix-Matrix Subroutines

Level 3: matrix-matrix subroutines include:

SGEMM, **DGEMM**, **CGEMM**, **ZGEMM**	Perform matrix-matrix operations on general matrices
SSYMM, **DSYMM**,**CSYMM**, **ZSYMM**	Perform matrix-matrix operations on symmetrical matrices
CHEMM,**ZHEMM**	Perform matrix-matrix operations on Hermitian matrices
SSYRK, **DSYRK**,**CSYRK**, **ZSYRK**	Perform symmetric rank k operations
CHERK, **ZHERK**	Perform Hermitian rank k operations
SSYR2K, **DSYR2K**, **CSYR2K**, **ZSYR2K**	Perform symmetric rank 2k operations
CHER2K,**ZHER2K**	Perform Hermitian rank 2k operations
STRMM, **DTRMM**,**CTRMM**, **ZTRMM**,	Perform matrix-matrix operations on triangular matrixes
STRSM, **DTRSM**, **CTRSM**, **ZTRSM**	Solve certain matrix equations

List of Numerical Manipulation Subroutines

These functions perform numerical manipulation:

a64l, **l64a**	Convert between long integers and base-64 ASCII strings
abs, **div**, **labs**, **ldiv**, **imul_dbl**, **umul_dbl, llabs, lldiv**	Compute absolute value, division, and multiplication of integers
asin, asinl, acos, acosl, atan, atanl, atan2, atan2l	Compute inverse trigonometric functions
asinh, **acosh**, **atanh**	Compute inverse hyperbolic functions
atof, **atoff**, **strtod**, **strtold**, **strtof**	Convert an ASCII string to a floating point number
bessell: j0, **j1**, **jn**, **y0**, **y1**, **yn**	Compute bessel functions
class, **finite**, **isnan**, **unordered**	Determine types of floating point functions
copysign, **nextafter**, **scalb**, **logb**, **ilogb**	Compute certain binary floating-point functions
nrand48, **mrand48**, **jrand48**, **srand48**, **seed48**, **lcong48**	Generate pseudo-random sequences
lrand48_r, **mrand48_r**, **nrand48_r**, **seed48_r**, or **srand48_r**	Generate pseudo-random sequences
drem or **remainder**	Compute an IEEE remainder
ecvt, **fcvt**, **gcvt**	Convert a floating-point number to a string
erf, **erfl**, **erfc**, **erfcl**	Compute error and complementary error functions
exp, **expl**, **expm1**, **log**, **logl**, **log10**, **log10l**, **log1p**, **pow**, **powl**	Compute exponential, log, and power functions
floor, **floorl**, **ceil**, **ceill**, **nearest**,	

trunc, **rint**, **itrunc**, **uitrunc**, **fmod**, **fmodl**, **fabs**, **fabsl**	Round floating-point numbers
fp_any_enable, **fp_is_enabled**, **fp_enable_all**, **fp_enable**, **fp_disable_all**, **fp_disable**	Allow operations on the floating-point exception status
fp_clr_flag, **fp_set_flag**, **fp_read_flag**, or **fp_swap_flag**	Allow operations on the floating-point exception status
fp_invalid_op, **fp_divbyzero**, **fp_overflow**, **fp_underflow**, **fp_inexact**, **fp_any_xcp**	Test to see if a floating-point exception has occurred
fp_iop_snan, **fp_iop_infsinf**, **fp_iop_infdinf**, **fp_iop_zrdzr**, **fp_iop_infmzr**, **fp_iop_invcmp**	Test to see if a floating-point exception has occurred
fp_read_rnd,**fp_swap_rnd**	Read and set the IEEE rounding mode
frexp, **frexpl**, **ldexp**, **ldexpl**, **modf**, **modfl**	Manipulate floating point numbers
l64a_r	Converts base-64 long integers to strings
lgamma, **lgammal**, **gamma**	Compute the logarithm of the gamma function
hypot, **cabs**	Compute Euclidean distance functions and absolute values
l3tol, **ltol3**	Convert between 3-byte integers and long integers
madd, **msub**, **mult**, **mdiv**, **pow**, **gcd**, **invert**, **rpow**, **msqrt**, **mcmp**, **move**, **min**, **omin**, **fmin**, **m_in**, **mout**, **omout**, **fmout**, **m_out**, **sdiv**, **itom**	Provide multiple precision integer arithmetic
rand, **srand**	Generate random numbers
rand_r	Generates random numbers
random, **srandom**, **initstate**, **setstate**	Generate better random numbers
rsqrt	Computes the reciprocal of the square root of a number
sin, **cos**, **tan**	Compute trigonometric and inverse trigonometric functions
sinh, **sinhl**, **cosh**, **coshl**, **tanh**, **tanhl**	Computes hyperbolic functions
sqrt, **sqrtl**, **cbrt**	Compute square root and cube root functions
strtol, **strtoll**, **strtoul**, **strtoull**, **atol**, **atoi**	Convert a string to an integer

List of Long Long Integer Numerical Manipulation Subroutines

The following subroutines perform numerical manipulation of integers stored in the long long integer data format:

llabs	Computes the absolute value of a long long integer
lldiv	Computes the quotient and remainder of the division of two long long integers
strtoll	Converts a string to a signed long long integer
strtoull	Converts a string to an unsigned long long integer
wcstoll	Converts a wide character string to a signed long long integer
wcstoull	Converts a wide character string to an unsigned long long integer

List of 128-Bit Long Double Numerical Manipulation Subroutines

The following subroutines perform numerical manipulation of floating-point numbers stored in the 128-bit long double data type. These subroutines do not support the 64-bit long double data type. Applications that use the 64-bit long double data type should use the corresponding double-precision subroutines.

acosl	Computes the inverse cosine of a floating-point number in long double format
asinl	Computes the inverse sine of a floating-point number in long double format
atan2l	Computes the principal value of the arc tangent of *x*/*y*, whose components are expressed in long double format
atanl	Computes the inverse tangent of a floating-point number in long double format
ceill	Computes the smallest integral value not less than a specified floating-point number in long double format
coshl	Computes the hyperbolic cosine of a floating-point number in long double format
cosl	Computes the cosine of a floating-point number in long double format

erfcl	Computes the value of 1 minus the error function of a floating-point number in long double format
erfl	Computes the error function of a floating-point number in long double format
expl	Computes the exponential function of a floating-point number in long double format
fabsl	Computes the absolute value of a floating-point number in long double format
floorl	Computes the largest integral value not greater than a specified floating-point number in long double format
fmodl	Computes the long double remainder of a fraction *x*/*y*, where *x* and *y* are floating-point numbers in long double format
frexpl	Expresses a floating-point number in long double format as a normalized fraction and an integral power of 2, storing the integer and returning the fraction
ldexpl	Multiplies a floating-point number in long double format by an integral power of 2
lgammal	Computes the natural logarithm of the absolute value of the gamma function of a floating-point number in long double format
log10l	Computes the base 10 logarithm of a floating-point number in long double format
logl	Computes the natural logarithm of a floating-point number in long double format
modfl	Stores the integral part of a real number in a long double variable and returns the fractional part of the real number
powl	Computes the value of *x* raised to the power of *y*, where both numbers are floating-point numbers in long double format
sinhl	Computes the hyperbolic sine of a floating-point number in long double format
sinl	Computes the sine of a floating-point number in long double format
sqrtl	Computes the square root of a floating-point number in long double format
strtold	Converts a string to a floating-point number in long double format
tanl	Computes the tangent of a floating-point number in long double format
tanhl	Computes the hyperbolic tangent of a floating-point number in long double format

List of Processes Subroutines

With the introduction of threads in the operating system, several process subroutines have been extended and other subroutines have been added. Threads, not processes, are now the schedulable entity. For signals, the handler exists at the process level, but each thread can define a signal mask. Some examples of changed or new subroutines are: **getprocs**, **getthrds**, **ptrace**, **getpri**, **setpri**, **yield** and **sigprocmask**.

Process Initiation

exec:, **execl**, **execv**, **execle**, **execve**, **execlp**, **execvp**, or **exect**	Execute new programs in the calling process
fork or **vfork**	Create a new process
reboot	Restarts the system
siginterrupt	Sets subroutines to restart when they are interrupted by specific signals

Process Suspension

pause	Suspends a process until that process receives a signal
wait, **wait3**, **waitpid**	Suspend a process until a child process stops or terminates

Process Termination

abort	Terminates current process and produces a memory dump by sending a **SIGOT** signal
exit, **atexit**, or **_exit**	Terminate a process
kill or **killpg**	Terminate current process or group of processes with a signal

Process and Thread Identification

ctermid	Gets the path name for the terminal that controls the current process
cuserid	Gets the alphanumeric user name associated with the current process
getpid, **getpgrp**, or **getppid**	Get the process ID, process group ID, or the parent process ID, respectively
getprocs	Gets process table entries
getthrds	Gets thread table entries
setpgid or **setpgrp**	Set the process group ID
setsid	Creates a session and sets process group IDs
uname or **unamex**	Gets the names of the current operating system

Process Accounting

acct	Enables and disables process accounting
ptrace	Traces the execution of a process

Process Resource Allocation

brk or **sbrk**	Change data segment space allocation
getdtablesize	Gets the descriptor table size
getrlimit, **setrlimit**, or **vlimit**	Limit the use of system resources by current process
getrusage, **times**, or **vtimes**	Display information about resource use
plock	Locks processes, text, and data into memory
profil	Starts and stops program address sampling for execution profiling
ulimit	Sets user process limits

Process Prioritization

getpri	Returns the scheduling priority of a process
getpriority, **setpriority**, or **nice**	Get or set the priority value of a process
setpri	Sets a process scheduling priority to a constant value
yield	Yields the processor to processes with higher priorities

Process and Thread Synchronization

compare_and_swap	Conditionally updates or returns a single word variable atomically
fetch_and_add	Updates a single word variable atomically
fetch_and_and and **fetch_and_or**	Sets or clears bits in a single word variable atomically
semctl	Controls semaphore operations
semget	Gets a set of semaphores
semop	Performs semaphore operations

Process Signals and Masks

raise	Sends a signal to an executing program

sigaction, **sigvec**, or **signal**	Specifies the action to take upon delivery of a signal
sigemptyset, **sigfillset**, **sigaddset**, **sigdelset**, or **sigismember**	Create and manipulate signal masks
sigpending	Determines the set of signals that are blocked from delivery
sigprocmask, **sigsetmask**, or **sigblock**	Set signal masks
sigset, **sighold**, **sigrelse**, or **sigignore**	Enhance the signal facility and provide signal management
sigsetjmp or **siglongjmp**	Save and restore stack context and signal masks
sigstack	Sets signal stack context
sigsuspend	Changes the set of blocked signals
ssignal or **gsignal**	Implement a software signal facility

Process Messages

msgctl	Provides message control operations
msgget	Displays a message queue identifier
msgrcv	Reads messages from a queue
msgsnd	Sends messages to the message queue
msgxrcv	Receives an extended message
psignal	Printing system signal messages

List of Multi-threaded Programming Subroutines

Programming in a multithreaded environment requires reentrant subroutines to ensure data integrity. Use the following subroutines rather than the nonreentrant version:

asctime_r	Converts a time value into a character array
getgrnam_r	Returns the next group entry in the user database that matches a specific name
getpwuid_r	Returns the next entry that matches a specific user ID in the use database

List of Programmer's Workbench Library Subroutines

The Programmers Workbench Library (**libPW.a**) contains routines that are provided only for compatibility with existing programs. Their use in new programs is not recommended. These interfaces are from AT&T PWB Toolchest.

any (*Character, String***)**	Determines whether *String* contains *Character*
anystr (*String1, String2***)**	Determines the offset in *String1* of the first character that also occurs in *String2*
balbrk (*String, Open, Close, End***)**	Determines the offset in *String* of the first character in the string *End* that occurs outside of a balanced string as defined by *Open* and *Close*
cat (*Destination, Source1, Source0***)**	Concatenates the *Source* strings and copies them to *Destination*
clean_up ( )	Defaults the cleanup routine
curdir (*String***)**	Puts the full path name of the current directory in *String*
dname (*p***)**	Determines which directory contains the file *p*
fatal (*Message***)**	General purpose error handler
fdfopen (*fd, Mode***)**	Same as the **stdio fdopen** subroutine
giveup (*Dump***)**	Forces a core dump
imatch (*pref, String***)**	Determines if the string *pref* is an initial substring of *String*

lockit (*LockFile, Count, pid***)**	Creates a lock file
move (*String1, String2, n***)**	Copies the first *n* characters of *String1* to *String2*
patoi (*String***)**	Converts *String* to integer
patol (*String***)**	Converts *String* to long**.**
repeat (*Destination, String, n***)**	Sets *Destination* to *String* repeated *n* times
repl (*String, Old, New***)**	Replaces each occurrence of the character *Old* in *String* with the character *New*
satoi (*String, *ip***)**	Converts *String* to integer and saves it in **ip*
setsig ( )	Causes signals to be caught by **setsig1**
setsig1 (*Signal***)**	General purpose signal handling routine
sname (*String***)**	Gets a pointer to the simple name of full path name *String*
strend (*String***)**	Finds the end of *String*.
trnslat (*s, old, new, Destination***)**	Copies string *s* into *Destination* and replace any character in *old* with the corresponding characters in *new*
unlockit (*lockfile, pid***)**	Deletes the lock file
userdir (*uid***)**	Gets the user's login directory
userexit (*code***)**	Defaults user exit routine
username (*uid***)**	Gets the user's login name
verify (*String1, String2***)**	Determines the offset in string *String1* of the first character that is not also in string *String2*
xalloc (*asize***)**	Allocates memory
xcreat (*name, mode***)**	Creates a file
xfree (*aptr***)**	Frees memory
xfreeall ( )	Frees all memory
xlink (*f1, f2***)**	Links files
xmsg (*file, func***)**	Calls the routine **fatal** with an appropriate error message
xpipe (*f***)**	Creates a pipe
xunlink (*f***)**	Removes a directory entry
xwrite (*fd, buffer, n***)**	Writes *n* bytes to the file associated with *fd* from *buffer*
zero (*p, n***)**	Zeros *n* bytes starting at address *p*
zeropad (*s***)**	Replaces the initial blanks with the character 0 (zero) in string *s*

File

/usr/lib/libPW.a	Contains routines provided only for compatibility with existing programs

List of Security and Auditing Subroutines

Access Control Subroutines

acl_chg or **acl_fchg**	Change the access control information on a file
acl_get or **acl_fget**	Get the access control information of a file
acl_put or **acl_fput**	Set the access control information of a file
acl_set or **acl_fset**	Set the base entries of the access control information of a file
chacl or **fchac l**	Change the permissions on a file
chmod or **fchmod**	Change file access permissions
chown, **fchown**, **chownx**, or **fchownx**	Change file ownership
frevoke	Revokes access to a file by other processes

revoke	Revokes access to a file
statacl or **fstatacl**	Retrieve the access control information for a file

Auditing Subroutines

audit	Enables and disables system auditing
auditbin	Defines files to contain audit records
auditevents	Gets or sets the status of system event auditing
auditlog	Appends an audit record to an audit bin file
auditobj	Gets or sets the auditing mode of a system data object
auditpack	Compresses and uncompresses audit bins
auditproc	Gets or sets the audit state of a process
auditread or **auditread_r**	Read an audit record
auditwrite	Writes an audit record

Identification and Authentication Subroutines

User authentication routines have a potential to store passwords and encrypted passwords in memory. This may expose passwords and encrypted passwords in coredumps.

authenticate	Authenticates the user's name and password
ckuseracct	Checks the validity of a user account
ckuserID	Authenticates the user
crypt, **encrypt**, or **setkey**	Encrypt or decrypt data
getgrent, **getgrgid**, **getgrnam**, **setgrent**, or **endgrent**	Access the basic group information in the user database
getgrgid_r	Gets a group database entry for a group ID in a multithreaded environment
getgrnam_r	Searches a group database for a name in a multithreaded environment
getgroupattr, **IDtogroup**, **nextgroup**, or **putgroupattr**	Access the group information in the user database
getlogin	Gets the user's login name
getlogin_r	Gets the user's login name in a multithreaded environment
getpass	Reads a password
getportattr or **putportattr**	Access the port information in the port database
getpwent, **getpwuid**, **getpwnam**, **putpwent**, **setpwent**, or **endpwent**	Access the basic user information in the user database
getuinfo	Finds the value associated with a user
getuserattr, **IDtouser**, **nextuser**, or **putuserattr**	Access the user information in the user database
getuserpw, putuserpw, or **putuserpwhist**	Access the user authentication data
loginfailed	Records an unsuccessful login attempt
loginrestrictions	Determines if a user is allowed to access the system
loginsuccess	Records a successful login
newpass	Generates a new password for a user
passwdexpired	Checks the user's password to determine if it has expired
setpwdb or **endpwdb**	Open or close the authentication database
setuserdb or **enduserdb**	Open or close the user database
system	Runs a shell command
tcb	Alters the Trusted Computing Base status of a file

Process Subroutines

getgid or **getegid**	Get the real or group ID of the calling process
getgroups	Gets the concurrent group set of the current process
getpcred	Gets the current process security credentials

getpenv	Gets the current process environment
getuid or **geteuid**	Get the real or effective user ID of the current process
initgroups	Initializes the supplementary group ID of the current process
kleenup	Cleans up the run-time environment of a process
setgid, **setrgid**, **setegid**, or **setregid**	Set the group IDs of the calling process
setgroups	Sets the supplementary group ID of the current process
setpcred	Sets the current process credentials
setpenv	Sets the current process environment
setuid, **setruid**, **setuid**, or **setreuid**	Set the process user IDs
usrinfo	Gets and sets user information about the owner of the current process

List of String Manipulation Subroutines

The string manipulation functions include:

- Locate a character position within a string
- Locate a sequence of characters within a string
- Copy a string
- Concatenate strings
- Compare strings
- Translate a string
- Measure a string

When using these string functions, you do not need to include a header file for them in the program or specify a special flag to the compiler.

The following functions manipulate string data:

bcopy, **bcmp**, **bzero**, **ffs**	Perform bit and byte string operations
gets, **fgets**	Get a string from a stream
puts, **fputs**	Write a string to a stream
compile, **step**, **advance**	Compile and match regular-expression patterns
strlen, **strchr**, **strrchr**, **strpbrk**, **strspn**, **strcspn**, **strstr**, **strtok**	Perform operations on strings
jcode	Performs string conversion on 8-bit processing codes.
varargs	Handles a variable-length parameter list

Programming Example for Manipulating Characters

```
/*
This program is designed to demonstrate the use of "Character
classification and conversion" subroutines. Since we are dealing
with characters, it is a natural place to demonstrate the use of
getchar subroutine and putchar subroutine from the stdio library.

The program objectives are:
-Read input from "stdin"
-Verify that all characters are ascii and printable
-Convert all uppercase characters to lowercase
-Discard multiple white spaces
-Report statistics regarding the types of characters
```

```
The following routines are demonstrated by this example program:

- getchar

- putchar

- isascii (ctype)

- iscntrl (ctype)

- isspace (ctype)

- isalnum (ctype)

- isdigit (ctype)

- isalpha (ctype)

- isupper (ctype)

- islower (ctype)

- ispunct (ctype)

- tolower (conv)

- toascii ( conv)
*/

#include <stdio.h>  /* The mandatory include file  */
#include <ctype.h>  /* Included for character classification
subroutines */

/* The various statistics gathering counters */
int asciicnt, printcnt, punctcnt, uppercnt, lowercnt,
digcnt, alnumcnt, cntrlcnt, spacecnt, totcnt, nonprntcnt,linecnt, tabcnt ;

main()
{

int ch ; /* The input character is read in to this */
char c , class_conv() ;

asciicnt=printcnt=punctcnt=uppercnt=lowercnt=digcnt==0;
cntrlcnt=spacecnt=totcnt=nonprntcnt=linecnt=tabcnt=0;
alnumcnt=0;

while ( (ch =getchar()) != EOF )
{

totcnt++;
c = class_conv(ch) ;
putchar(c);
```

```
}
printf("The number lines of of input were %d\n",linecnt);
printf(" The character wise breakdown follows :\n");
printf(" TOTAL  ASCII  CNTRL   PUNCT   ALNUM  DIGITS UPPER
LOWER   SPACE   TABCNT\n");

printf("%5d %5d %5d %5d %5d %5d %5d %5d %5d %5d\n",totcnt,
asciicnt, cntrlcnt, punctcnt, alnumcnt, digcnt, uppercnt,lowercnt, spacecnt, tabcnt );

}

char class_conv(ch)
char ch;
{

if (isascii(ch)) {

asciicnt++;
if ( iscntrl(ch) && ! isspace(ch)) {

nonprntcnt++ ;
cntrlcnt++ ;
return(' ');

}
else if ( isalnum(ch)) {

alnumcnt++;
if (isdigit(ch)){
digcnt++;
return(ch);
}
else if (isalpha(ch)){
if ( isupper(ch) ){
uppercnt++ ;
return(tolower(ch));
}
else if ( islower(ch) ){
lowercnt++;
return(ch);
}
else {
/*
We should never be in this situation since an alpha character can only be
either uppercase or lowercase.
*/
fprintf(stderr,"Classification error for %c \n",ch);
return(NULL);
}
}
else if (ispunct(ch) ){
punctcnt++;
return(ch);
}
else if ( isspace(ch) ){
```

```
spacecnt++;
if ( ch == '\n' ){
linecnt++;
return(ch);
}
while ( (ch == '\t' ) || ( ch == ' ' ) ) {
if ( ch == '\t' ) tabcnt ++ ;
else if ( ch == ' ' ) spacecnt++ ;
totcnt++;
ch = getchar();
}
ungetc(ch,stdin);
totcnt--;
return(' ');
}
else {
/*
We should never be in this situation any ASCII character
can only belong to one of the above classifications.
*/
fprintf(stderr,"Classification error for %c \n",ch);
return(NULL);
}
}
else
{
fprintf(stdout,"Non Ascii character encountered \n");
return(toascii(ch));
}
}
```

Searching and Sorting Example Program

```
/**This program demonstrates the use of the following:
-qsort subroutine (a quick sort library routine)
-bsearch subroutine (a binary search library routine)
-fgets, fopen, fprintf, malloc, sscanf, and strcmp subroutines.
The program reads two input files with records in
string format, and prints or displays:

-records from file2, which are excluded in file1

-records from file1, which are excluded in file2

The program reads the input records from both files
into two arrays, which are  subsequently sorted in
common order using the qsort subroutine. Each element of
one array is searched for its counterpart entry in the
other array using the bsearch subroutine. If the item is
not found in both arrays, a message indicates the record
was not found. The process is repeated interchanging
the two arrays, to obtain the second list of exclusions.

**/

#include <stdio.h>      /*the library file to be included for
                         /*standard input and output*/
#include <search.h>     /*the file to be included for qsort*/
#include <sys/errno.h>  /*the include file for interpreting
```

```
                        /*predefined error conditions*/
#define MAXRECS  10000           /*array size limit*/
#define MAXSTR   256             /*maximum input string length*/
#define input1 "file1"           /*one input file*/
#define input2 "file2"           /*second input file*/
#define out1   "o_file1"         /*output file1*/
#define out2   "o_file2"         /*output file2*/

main()
{
char *arr1[MAXRECS] , *arr2[MAXRECS] ;/*the arrays to store
input records*/
unsigned int num1 , num2;        /*to keep track of the number of
                                 /*input records. Unsigned int
                                 /*declaration ensures
                                 /*compatability
                                 /*with qsort library routine.*/
int i ;
int compar();             /*the function used by qsort and
                          /*bsearch*/
extern int errno ; /*to capture system call failures*/
FILE *ifp1 , *ifp2, *ofp1, *ofp2; /*the file pointers for
input and output */
void *bsearch() ;        /*the library routine for binary search*/
void qsort();            /*the library routine for quick sort*/
char*malloc() ;          /*memory allocation subroutine*/
void exit() ;

num1 = num2 = 0;

/**Open the input and output files for reading or writing
**/

if ( (ifp1 = fopen( input1 , "r" ) ) == NULL )
{
(void) fprintf(stderr,"%s could not be opened\n",input1);
exit(-1);
}
if (( ifp2 = fopen( input2 , "r" )) == NULL )
{
(void) fprintf(stderr,"%s could not be opened\n",input2);
exit(-1);
}
if (( ofp1 = fopen(out1,"w" )) == NULL )
{
(void) fprintf(stderr,"%s could not be opened\n",out1);
exit(-1);
}
if (( ofp2 = fopen(out2,"w")) == NULL )
{
(void) fprintf(stderr,"%s could not be opened\n", out2);
exit(-1);
}
/**Fill the arrays  with data from input files. Readline
function returns the number of input records.**/
```

```
if ( (i = readline( arr1 , ifp1 ))  < 0 )
{
(void) fprintf(stderr,"o data in %s. Exiting\n",input1);
exit(-1);
}
num1 = (unsigned) i;
if ( (i = readline ( arr2 , ifp2)) < 0 )
{
(void) fprintf(stderr,"No data in %s.  Exiting\n",input2);
exit(-1);
}
num2 = (unsigned) i;
/**
The arrays can now be sorted using qsort subroutine
**/
qsort( (char *)arr1 , num1 ,  sizeof (char *)  , compar);
qsort( (char *)arr2 , num2 ,  sizeof (char *)  , compar);
/**When the two arrays are sorted in a common order, the
program builds a list of elements found in one but not
in the other, using bsearch.
Check that each element in array1 is in array2
**/
for ( i= 0 ; i < num1 ; i++ )
{
if ( bsearch((void *)&arr1[i] , (char *)arr2,num2,
sizeof(char *) , compar) == NULL )
{
(void)  fprintf(ofp1,"%s",arr1[i]);
}
} /**One list of exclusions is complete**/
/**Check that each element in array2 is in array1**/
for ( i = 0 ; i < num2 ; i++ )
{
 if ( bsearch((void *)&arr2[i], (char *)arr1, num1
, sizeof(char *) , compar) == NULL )
{
(void) fprintf(ofp2,"%s",arr2[i]);
}
}
/**Task completed, so return**/
return(0);
}
/**The function reads in records from an input
 file and fills in the details into the two arrays.**/
readline ( char **aptr, FILE *fp )
{
char str[MAXSTR] , *p ;
int i=0 ;
/**Read the input file line by line**/
while ( fgets(str , sizeof(str) , fp ))
{
/**Allocate sufficient memory. If the malloc subroutine
fails, exit.**/
if ( (p = (char *)malloc ( sizeof(str))) == NULL )
{
```

```
(void) fprintf(stderr,"Insufficient Memory\n");
return(-1);
}
else
{
if ( 0 > strcpy(p, str))
{
(void) fprintf(stderr,"Strcpy failed \n");
return(-1);
}
i++ ; /*increment number of records count*/
}
} /**End of input file reached**/
return(i);/*return the number of records read*/
}
/**We want to sort the arrays based only on the contents of the first field of
the input records. So we get the first  field using SSCANF**/
compar( char **s1 , char **s2 )
{
char st1[100] , st2[100] ;
(void) sscanf(*s1,"%s" , st1) ;
(void) sscanf(*s2,"%s" , st2) ;
/**Return the results of string comparison to the calling procedure**/
return(strcmp(st1 , st2));
}
```

List of Operating System Libraries

/usr/lib/libbsd.a	Berkeley library
/lib/profiled/libbsd.a	Berkeley library profiled
/usr/ccs/lib/libcurses.a	Curses library
/usr/ccs/lib/libc.a	Standard I/O library, standard C library
/lib/profiled/libc.a	Standard I/O library, standard C library profiled
/usr/ccs/lib/libdbm.a	Database Management library
/usr/ccs/lib/libi18n.a	Layout library
/usr/lib/liblvm.a	LVM (Logical Volume Manager) library
/usr/ccs/lib/libm.a	Math library
/usr/ccs/lib/libp/libm.a	Math library profiled
/usr/lib/libodm.a	ODM (Object Data Manager) library
/usr/lib/libPW.a	Programmers Workbench library
/usr/lib/libpthreads.a	POSIX compliant Threads library
/usr/lib/libqb.a	Queue Backend library
/usr/lib/librpcsvc.a	RPC (Remote Procedure Calls) library
/usr/lib/librts.a	Run-Time Services library
/usr/lib/libs.a	Security functions
/usr/lib/libsm.a	System management library
/usr/lib/libsrc.a	SRC (System Resource Controller) library
/usr/lib/libmsaa.a	SVID (System V Interface Definition) math library
/usr/ccs/lib/libp/libmsaa.a	SVID (System V Interface Definition) math library profiled
/usr/ccs/lib/libtermcap.a	Terminal I/O
/usr/lib/liby.a	YP (Yellow Pages) library
/usr/lib/lib300.a	Graphics subroutines for DASI 300 workstations
/usr/lib/lib300s.a	Graphics subroutines for DASI 300s workstations
/usr/lib/lib300S.a	Graphics subroutines for DASI 300S workstations
/usr/lib/lib4014.a	Graphics subroutines for Tektronix 4014 workstations

/usr/lib/lib450.a	Graphics subroutines for DASI 450 workstations
/usr/lib/libcsys.a	Kernel extensions services
/usr/ccs/lib/libdbx.a	Debug program library
/usr/lib/libgsl.a	Graphics Support library
/usr/lib/libieee.a	IEEE floating point library
/usr/lib/liblM.a	Stanza file processing library
/usr/ccs/lib/libl.a	lex library
/usr/lib/libogsl.a	Old graphics support library
/usr/lib/liboldX.a	X10 library
/usr/lib/libplot.a	Plotting subroutines
/usr/lib/librpcsvc.a	RPC services
/usr/lib/librs2.a	Hardware-specific **sqrt** and **itrunc** subroutines
/usr/lib/libxgsl.a	Enhanced X-Windows graphics subroutines
/usr/lib/libX11.a	X11 run time library
/usr/lib/libXt.a	X11 toolkit library
/usr/lib/liby.a	yacc run time library

librs2.a Library

The **/usr/lib/librs2.a** library provides statically linked, hardware-specific replacements for the **sqrt** and **itrunc** subroutines. These replacement subroutines make use of hardware-specific instructions on some POWER-based, POWERstation, and POWERserver models to increase performance.

> **Note:** Use the hardware-specific versions of these subroutines in programs that will run only on models of POWER-based machines, POWERstations, and POWERservers that support hardware implementations of square root and conversion to integer. Attempting to use them in programs running on other models will result in an illegal instruction trap.

General-Use sqrt and itrunc Subroutines

The general-use version of the **sqrt** subroutine is in the **libm.a** library. The **sqrt** subroutine computes the square root of a floating-point number.

The general-use version of the **itrunc** subroutine is in the **libc.a** library. The **itrunc** subroutine converts a floating-point number to integer format.

POWER2-Specific sqrt and itrunc Subroutines

The **/usr/lib/librs2.a** library contains the following subroutines:

- **sqrt**
- **_sqrt**
- **itrunc**
- **_itrunc**

The subroutine names with leading underscores are used by the C and Fortran compilers. They are functionally identical to the versions without underscores.

For best performance, source code that computes square roots or converts floating-point numbers to integers can be recompiled with the **xlc** or **xlf** command using the **-qarch=pwrx** compiler option. This option enables a program to use the square-root and convert-to-integer instructions.

To use the hardware-specific subroutines in the **librs2.a** library, link it ahead of the **libm.a** and **libc.a** libraries. For example:

```
xlc -O -o prog prog.c -lrs2 -lm
```

```
OR
xlf -O -o prog prog.f -lrs2
```

You can use the **xlf** or **xlc** compiler to rebind a program to use this library. For example, to create a POWER2-specific executable file named `progrs2` from an existing non-stripped file named `prog` in the current directory:

```
xlc -lrs2 prog -o progrs2
OR
xlf -lrs2 prog -o progrs2
```

Chapter 25. System Management Interface Tool (SMIT)

The System Management Interface Tool (SMIT) is an interactive and extensible screen-oriented command interface. It prompts users for the information needed to construct command strings and presents appropriate predefined selections or run time defaults where available. This shields users from many sources of extra work or error, including the details of complex command syntax, valid parameter values, system command spelling, or custom shell path names.

You can also build and use alternate databases instead of modifying SMIT's default system database.

The following sections discuss SMIT in detail:

New tasks consisting of one or more commands or inline **ksh** shell scripts can be added to SMIT at any time by adding new instances of predefined screen objects to SMIT's database. These screen objects (described by stanza files) are used by the Object Data Manager (ODM) to update SMIT's database. This database controls SMIT's run-time behavior.

SMIT Screen Types

There are three main screen types available for the System Management Interface Tool (SMIT). The screens occur in a hierarchy consisting of menu screens, selector screens, and dialog screens. When performing a task, a user typically traverses one or more menus, then zero or more selectors, and finally one dialog.

The following table shows SMIT screen types, what the user sees on each screen, and what SMIT does internally with each screen:

SMIT Screens		
Screen Type	**What the User Sees on the Screen**	**What SMIT Does Internally with Each Screen**
Menu	A list of choices	Uses the choice to select the next screen to display.
Selector	Either a list of choices or an entry field	Obtains a data value for subsequent screens. Optionally selects alternative dialogs or selectors.
Dialog	A sequence of entry fields.	Uses data from the entry fields to construct and run the target task command string.

Menus present a list of alternative subtasks; a selection can then lead to another menu screen or to a selector or dialog screen. A selector is generally used to obtain one item of information that is needed by a subsequent screen and which can also be used to select which of several selector or dialog screens to use next. A dialog screen is where any remaining input is requested from the user and where the chosen task is actually run.

A menu is the basic entry point into SMIT and can be followed by another menu, a selector, or a dialog. A selector can be followed by a dialog. A dialog is the final entry panel in a SMIT sequence.

Menu Screens

A SMIT menu is a list of user-selectable items. Menu items are typically tasks or classes of tasks that can be performed from SMIT. A user starting with the main SMIT menu selects an item defining a broad range

of system tasks. A selection from the next and subsequent menus progressively focuses the user's choice, until finally a dialog is typically displayed to collect information for performance of a particular task.

Design menus to help a user of SMIT narrow the scope of choice to a particular task. Your design can be as simple as a new menu and dialog attached to an existing branch of SMIT, or as complex as an entire new hierarchy of menus, selectors, and dialogs starting at the SMIT applications menu.

At run time, SMIT retrieves all menu objects with a given ID (**id** descriptor value) from the specified object repository. To add an item to a particular SMIT menu, add a menu object having an ID value equal to the value of the **id** descriptor of other non-title objects in the same menu.

Build menus by defining them in a stanza file and then processing the file with the **odmadd** command. A menu definition is compiled into a group of menu objects. Any number of menus, selectors, and dialogs can be defined in one or more files.

odmadd	Adds the menu definitions to the specified object repository.
/usr/lib/objrepos	Default object repository for system information and can be used to store your compiled objects.

At SMIT run time, the objects are automatically retrieved from a SMIT database.

> **Note:** You should always back up the **/usr/lib/objrepos** directory before deleting or adding any objects or object classes. Unanticipated damage to objects or classes needed for system operations can cause system problems.

Selector Screens

A SMIT selector prompts a user to specify a particular item, typically a system object (such as a printer) or attribute of an object (such as a serial or parallel printer mode). This information is then generally used by SMIT in the next dialog.

For instance, a selector can prompt a user to enter the name of a logical volume for which to change logical volume characteristics. This could then be used as a parameter in the `sm_cmd_hdr.cmd_to_discover_postfix` field of the next dialog for entry field initialization. Likewise, the selector value could also be used as the value for a subsequent `sm_cmd_opt.cmd_to_list_postfix` field. It can also be used directly as a subsequent initial entry field value. In each case, logical consistency requires that this item either be selected prior to the dialog or be held constant while in the dialog.

Design a selector to request a single piece of information from the user. A selector, when used, falls between menus and dialogs. Selectors can be strung together in a series to gather several pieces of information before a dialog is displayed.

Selectors should usually contain a prompt displayed in user-oriented language and either a response area for user input or a pop-up list from which to select a value; that is, one question field and one answer. Typically the question field is displayed and the SMIT user enters a value in the response area by typing the value or by selecting a value from a list or an option ring.

To give the user a run-time list of choices, the selector object can have an associated command (defined in the `sm_cmd_opt.cmd_to_list` field) that lists the valid choices. The list is not hard-coded, but developed by the command in conjunction with standard output. The user gets this list by selecting the **F4=List** function of the SMIT interface.

In a ghost selector (`sm_cmd_hdr.ghost="y"`), the command defined in the `sm_cmd_opt.cmd_to_list` field, if present, is automatically run. The selector screen is not displayed at this time and the user sees only the pop-up list.

The application of a super-ghost selector permits branching following menu selection, where the branch to be taken depends on the system state and not user input. In this case, the **cmd_to_classify** descriptor in the super-ghost selector can be used to get the required information and select the correct screen to present next.

Build selectors by defining them in a stanza file and then processing the file with the **odmadd** command. Several menus, selectors, and dialogs can be defined in a single file. The **odmadd** command adds each selector to the specified object repository. The **/usr/lib/objrepos** directory is the default object repository for system information and is used to store your compiled objects. At SMIT run time, the objects are automatically retrieved from a SMIT database.

> **Note:** Always back up the **/usr/lib/objrepos** directory before deleting or adding any objects or object classes. Unanticipated damage to objects or classes needed for system operations can cause system problems.

Dialog Screens

A dialog in SMIT is the interface to a command or task a user performs. Each dialog executes one or more commands, shell functions, and so on. A command can be run from any number of dialogs.

To design a dialog, you need to know the command string you want to build and the command options and operands for which you want user-specified values. In the dialog display, each of these command options and operands is represented by a prompt displayed in user-oriented language and a response area for user input. Each option and operand is represented by a dialog command option object in the Object Data Manager (ODM) database. The entire dialog is held together by the dialog header object.

The SMIT user enters a value in the response area by typing the value, or by selecting a value from a list or an option ring. To give the user a run-time list of choices, each dialog object can have an associated command (defined in the `sm_cmd_opt.cmd_to_list` field) that lists the valid choices. The user gets this list by invoking the **F4=List** function of the SMIT interface. This causes SMIT to run the command defined in the associated `cmd_to_list` field and to use its standard output and **stderr** file for developing the list.

In a ghost dialog, the dialog screen is not displayed. The dialog runs as if the user had immediately pressed the dialog screen **Enter** key to run the dialog.

Build dialogs by defining them in a stanza file and then processing the file with the **odmadd** command. Several menus, selectors, and dialogs can be defined in a single file. The **odmadd** command adds each dialog definition to the specified object repository. The **/usr/lib/objrepos** directory is the default object repository for system information and can be used to store your compiled objects. At SMIT run time, the objects are automatically retrieved from a SMIT database.

> **Note:** Always back up the **/usr/lib/objrepos** directory before deleting or adding any objects or object classes. Unanticipated damage to objects or classes needed for system operations can cause system problems.

SMIT Object Classes

A System Management Interface Tool (SMIT) object class created with the Object Data Manager (ODM) defines a common format or record data type for all individual objects that are instances of that object class. Therefore a SMIT object class is basically a record data type and a SMIT object is a particular record of that type.

SMIT menu, selector, and dialog screens are described by objects that are instances of one of four object classes:

- **sm_menu_opt**

- **sm_name_hdr**
- **sm_cmd_hdr**
- **sm_cmd_opt**

The following table shows the objects used to create each screen type:

SMIT Classes		
Screen Type	**Object Class**	**Object's Use (typical case)**
Menu	**sm_menu_opt**	1 for title of screen
	sm_menu_opt	1 for first item
	sm_menu_opt	1 for second item
	...	...
	sm_menu_opt	1 for last item
Selector	**sm_name_hdr**	1 for title of screen and other attributes
	sm_cmd_opt	1 for entry field or pop-up list
Dialog	**sm_cmd_hdr**	1 for title of screen and command string
	sm_cmd_opt	1 for first entry field
	sm_cmd_opt	1 for second entry field
	...	...
	sm_cmd_opt	1 for last entry field

Each object consists of a sequence of named fields and associated values. These are represented in stanza format in ASCII files that can be used by the **odmadd** command to initialize or extend SMIT databases. Stanzas in a file should be separated with one or more blank lines.

> **Note:** Comments in an ODM input file (ASCII stanza file) used by the **odmadd** command must be alone on a line beginning with a # (pound sign) or an * (asterisk) in column one. Only an * (asterisk) comment can be on the same line as a line of the stanza, and must be after the descriptor value.

The following is an example of a stanza for an **sm_menu_opt** object:

```
sm_menu_opt:                    *name of object class
  id             = "top_menu"   *object's (menu screen) name
  id_seq_num     = "050"
  next_id        = "commo"      *id of objects for next menu screen
  text           = "Communications Applications & Services"
  text_msg_file  = ""
  text_msg_set   = 0
  text_msg_id    = 0
  next_type      = "m"          *next_id specified another menu
  alias          = ""
  help_msg_id    = ""
  help_msg_loc   = ""
  help_msg_base  = ""
  help_msg_book  = ""
```

The notation *ObjectClass.Descriptor* is commonly used to describe the value of the fields of an object. For instance, in the preceding **sm_menu_opt** object, the value of **sm_menu_opt.id** is `top_menu`.

See "sm_menu_opt (SMIT Menu) Object Class" on page 673 for a detailed explanation of each field in the **sm_menu_opt** object class.

The following is an example of a stanza for an **sm_name_hdr** object:

```
sm_name_hdr:              *---- used for selector screens
  id                      = ""  *the name of this selector screen
  next_id                 = ""  *next sm_name_hdr or sm_cmd_hdr
  option_id               = ""  *specifies one associated sm_cmd_opt
  has_name_select         = ""
  name                    = ""  *title for this screen
  name_msg_file           = ""
  name_msg_id             = 0
  type                    = ""
  ghost                   = ""
  cmd_to_classify         = ""
  cmd_to_classify_postfix = ""
  raw_field_name          = ""
  cooked_field_name       = ""
  next_type               = ""
  help_msg_id             = ""
  help_msg_loc            = ""
  help_msg_base = ""
  help_msg_book = ""
```

See the “sm_name_hdr (SMIT Selector Header) Object Class” on page 674 for a detailed explanation of each field in the **sm_name_hdr** object class.

The following is an example of a stanza for an **sm_cmd_hdr** object:

```
sm_cmd_hdr:                       *---- used for dialog screens
  id                      = ""    *the name of this dialog screen
  option_id     = "" *defines associated set of sm_cmd_opt objects
  has_name_select         = ""
  name                    = ""    *title for this screen
  name_msg_file           = ""
  name_msg_set            = 0
  name_msg_id             = 0
  cmd_to_exec             = ""
  ask                     = ""
  exec_mode               = ""
  ghost                   = ""
  cmd_to_discover         = ""
  cmd_to_discover_postfix          = ""
  name_size               = 0
  value_size              = 0
  help_msg_id             = ""
  help_msg_loc            = ""
  help_msg_base   = ""
  help_msg_book   = ""
```

See the “sm_cmd_hdr (SMIT Dialog Header) Object Class” on page 680 for a detailed explanation of each field in the **sm_cmd_hdr** object class.

The following is an example of a stanza for an **sm_cmd_opt** object:

```
sm_cmd_opt:               *---- used for selector and dialog screens
  id                      = ""      *name of this object
  id_seq_num    = ""      *"0" if associated with selector screen
  disc_field_name         = ""
  name                    = ""      *text describing this entry
  name_msg_file           = ""
  name_msg_set            = 0
  name_msg_id             = 0
  op_type                 = ""
  entry_type              = ""
  entry_size              = 0
  required                = ""
  prefix                  = ""
  cmd_to_list_mode        = ""
  cmd_to_list             = ""
```

```
cmd_to_list_postfix    = ""
multi_select           = ""
value_index            = 0
disp_values            = ""
values_msg_file        = ""
values_msg_set         = 0
values_msg_id          = 0
aix_values             = ""
help_msg_id            = ""
help_msg_loc           = ""
help_msg_base   = ""
help_msg_book   = ""
```

See “sm_cmd_opt (SMIT Dialog/Selector Command Option) Object Class” on page 677 for a detailed explanation of each field in the **sm_cmd_opt** object class.

All SMIT objects have an `id` field that provides a name used for looking up that object. The **sm_menu_opt** objects used for menu titles are also looked up using their `next_id` field. The **sm_menu_opt** and **sm_name_hdr** objects also have `next_id` fields that point to the `id` fields of other objects. These are how the links between screens are represented in the SMIT database. Likewise, there is an `option_id` field in **sm_name_hdr** and **sm_cmd_hdr** objects that points to the `id` fields of their associated **sm_cmd_opt** object(s).

Note: The **sm_cmd_hdr.option_id** object field is equal to each **sm_cmd_opt.id** object field; this defines the link between the **sm_cmd_hdr** object and its associated **sm_cmd_opt** objects.

Two or more dialogs can share common **sm_cmd_opt** objects since SMIT uses the ODM **LIKE** operator to look up objects with the same `sm_cmd_opt.id` field values. SMIT allows up to five IDs (separated by commas) to be specified in a `sm_cmd_hdr.option_id` field, so that **sm_cmd_opt** objects with any of five different `sm_cmd_opt.id` field values can be associated with the **sm_cmd_hdr** object.

The following table shows how the value of an `sm_cmd_hdr.option_id` field relates to the values of the `sm_cmd_opt.id` and `sm_cmd_opt.id_seq_num` fields.

Note: The values in the `sm_cmd_opt.id_seq_num` fields are used to sort the retrieved objects for screen display.

SMIT Objects		
IDs of Objects to Retrieve (sm_cmd_hdr.option_id)	**Objects Retrieved (sm_cmd_opt.id)**	**Display Sequence of Retrieved Objects (sm_cmd_opt.id_seq_num)**
″demo.[AB]″	″demo.A″	″10″
	″demo.B″	″20″
	″demo.A″	″30″
	″demo.A	″40″
″demo.[ACD]″	″demo.A″	″10″
	″demo.C″	″20″
	″demo.A″	″30″
	″demo.A″	″40″
	″demo.D″	″50″
″demo.X,demo.Y,demo.Z″	″demo.Y″	″20″
	″demo.Z″	″40″
	″demo.X″	″60″
	″demo.X″	″80″

The SMIT Database

SMIT objects are generated with ODM creation facilities and stored in files in a designated database. The default SMIT database consists of eight files:

- **sm_menu_opt**
- **sm_menu_opt.vc**
- **sm_name_hdr**
- **sm_name_hdr.vc**
- **sm_cmd_hdr**
- **sm_cmd_hdr.vc**
- **sm_cmd_opt**
- **sm_cmd_opt.vc**

The files are stored by default in the **/usr/lib/objrepos** directory. They should always be saved and restored together.

SMIT Aliases and Fast Paths

A System Management Interface Tool (SMIT) **sm_menu_opt** object can be used to define a fast path that, when entered with the **smit** command to start SMIT, can get a user directly to a specific menu, selector, or dialog; the alias itself is never displayed. Use of a fast path allows a user to bypass the main SMIT menu and other objects in the SMIT interface path to that menu, selector, or dialog. Any number of fast paths can point to the same menu, selector, or dialog.

An **sm_menu_opt** object is used to define a fast path by setting the `sm_menu_opt.alias` field to `"y"`. In this case, the **sm_menu_opt** object is used exclusively to define a fast path. The new fast path or alias name is specified by the value in the `sm_menu_opt.id` field. The contents of the `sm_menu_opt.next_id` field points to another menu object, selector header object, or dialog header object, depending on whether the value of the `sm_menu_opt.next_type` field is `"m"` (menu), `"n"` (selector), or `"d"` (dialog).

Every non alias **sm_menu_opt** object for a menu title (`next_type="m"`) should have a unique `sm_menu_opt.next_id` field value, since this field is automatically used as a fast path.

If you want two menu items to point to the same successor menu, one of the `next_id` fields should point to an alias, which in turn points to the successor menu.

Build aliases and fast paths by defining them in a stanza file and then processing the file with the **odmadd** command. Several menus, selectors, and dialogs can be defined in a single file. The **odmadd** command adds each alias definition to the specified object repository. The **/usr/lib/objrepos** directory is the default object repository for system information and can be used to store your compiled objects. At SMIT run time, the objects are automatically retrieved from a SMIT database.

> **Note:** You should always back up the **/usr/lib/objrepos** directory before deleting or adding any objects or object classes. Unanticipated damage to objects or classes needed for system operations can cause system problems.

SMIT Information Command Descriptors

The System Management Interface Tool (SMIT) can use several descriptors defined in its objects to get the information, such as current run time values, required to continue through the SMIT interface structure. Each of these descriptors is assigned some form of command string to run and retrieve the needed data.

The descriptors that can be set to a command for discovery of required information are:

- The **cmd_to_discover** descriptor that is part of the **sm_cmd_hdr** object class used to define a dialog header.
- The **cmd_to_classify** descriptor that is part of the **sm_name_hdr** object class used to define a selector header.
- The **cmd_to_list** descriptor that is part of the **sm_cmd_opt** object class used to define a selector option list associated with a selector or a dialog command option list associated with a dialog entry field.

SMIT executes a command string specified by a **cmd_to_list**, **cmd_to_classify**, or **cmd_to_discover** descriptor by first creating a child process. The standard error (strerr) and standard output of the child process are redirected to SMIT via pipes. SMIT next executes a **setenv(**″**ENV=**″**)** subroutine in the child process to prevent commands specified in the **$HOME/.env** file of the user from being run automatically when a new shell is invoked. Finally, SMIT calls the **execl** system subroutine to start a new **ksh** shell, using the command string as the **ksh -c** parameter value. If the exit value is not 0, SMIT notifies the user that the command failed.

SMIT makes the path names of the log files and the settings of the command line **verbose**, **trace**, and **debug** flags available in the shell environment of the commands it runs. These values are provided via the following environment variables:

- **_SMIT_LOG_FILE**
- **_SMIT_SCRIPT_FILE**
- **_SMIT_VERBOSE_FLAG**
- **_SMIT_TRACE_FLAG**
- **_SMIT_DEBUG_FLAG**

The presence or absence of the corresponding flag is indicated by a value of 0 or 1, respectively.

An easy way to view the current settings is to invoke the shell function after starting SMIT and then run the command string **env | grep _SMIT**.

All writes to the log files should be done as appends and should be immediately followed by flushes unless this occurs automatically.

The cmd_to_discover Descriptor

When SMIT puts up a dialog, it gets the **sm_cmd_hdr** (dialog header) object and its associated dialog body (one or more **sm_cmd_opt** objects) from the object repository. However, the **sm_cmd_opt** objects can also be initialized with current run time values. If the `sm_cmd_hdr.cmd_to_discover` field is not empty (″″), SMIT runs the command specified in the field to obtain current run time values.

Any valid **ksh** command string can be used as a **cmd_to_discover** descriptor value. The command should generate the following output format as its standard output:

```
#name_1:name_2: ... :name_n\n
value_1:value_2: ... :value_n
```

In the standard output of a command, the first character is always a `#` (pound sign). A `\n` (new line character) is always present to separate the name line from the value line. Multiple names and values are separated by `:` (colons). And any name or value can be an empty string (which in the output format appears as two colons with no space between them). SMIT maintains an internal current value set in this format that is used to pass name-value pairs from one screen to the next.

> **Note:** If the value includes a `:` (colon), the `:` must be preceded by `#!` (pound sign, exclamation point). Otherwise, SMIT reads the `:` (colon) as a field separator.

When SMIT runs a command specified in a `cmd_to_discover` field, it captures the `stdout` of the command and loads these name-value pairs (`name_1` and `value_1` `name_2` and `value_2`, and so on) into the **disp_values** and **aix_values** descriptors of the **sm_cmd_opt** (dialog command option) objects by matching each name to a **sm_cmd_opt.disc_field_name** descriptor in each **sm_cmd_opt** object.

For a **sm_cmd_opt** (dialog command option) object that displays a value from a preceding selector, the **disc_field_name** descriptor for the dialog command option object must be set to ″`_rawname`″ or ″`_cookedname`″ (or whatever alternate name was used to override the default name) to indicate which value to use. In this case, the **disc_field_name** descriptor of the **sm_cmd_opt** (dialog command option) object should normally be a no-entry field. If a particular value should always be passed to the command, the **required** descriptor for the **sm_cmd_opt** (dialog command option) object must be set to `y` (yes), or one of the other alternatives.

A special case of option ring field initialization permits the current value for a **cmd_to_discover** descriptor (that is, any name-value pair from the current value set of a dialog) of a ring entry field to specify which pre-defined ring value to use as the default or initial value for the corresponding entry field. At dialog initialization time, when a dialog entry field matches a name in the current value set of the dialog (via **sm_cmd_opt.disc_field_name**), a check is made to determine if it is an option ring field (**sm_cmd_opt.op_type =** ″**r**″) and if it has predefined ring values (**sm_cmd_opt.aix_values !=** ″″). If so, this set of option ring values is compared with the current value for **disc_field_name** from the current value set. If a match is found, the matched option ring value becomes the default ring value (**sm_cmd_opt.value_index** is set to its index). The corresponding translated value (**sm_cmd_opt.disp_values**), if available, is displayed. If no match is found, the error is reported and the current value becomes the default and only value for the ring.

In many cases, discovery commands already exist. In the devices and storage areas, the general paradigms of add, remove, change, and show exist. For example, to add (**mk**), a dialog is needed to solicit characteristics. The dialog can have as its discovery command the show (**ls**) command with a parameter that requests default values. SMIT uses the standard output of the show (**ls**) command to fill in the suggested defaults. However, for objects with default values that are constants known at development time (that is, that are not based on the current state of a given machine), the defaults can be initialized in the dialog records themselves; in this case, no **cmd_to_discover** is needed. The dialog is then displayed. When all fields are filled in and the dialog is committed, the add (**mk**) command is executed.

As another example, a change (**ch**) dialog can have as its discovery command a show (**ls**) command to get current values for a given instance such as a particular device. SMIT uses the standard output of the show (**ls**) command to fill in the values before displaying the dialog. The show (**ls**) command used for discovery in this instance can be the same as the one used for discovery in the add (**mk**) example, except with a slightly different set of options.

The cmd_to_*_postfix Descriptors

Associated with each occurrence of a **cmd_to_discover**, **cmd_to_classify**, or **cmd_to_list** descriptor is a second descriptor that defines the postfix for the command string defined by the **cmd_to_discover**, **cmd_to_classify**, or **cmd_to_list** descriptor. The postfix is a character string defining the flags and parameters that are appended to the command before it is executed.

The descriptors that can be used to define a postfix to be appended to a command are:

- The **cmd_to_discover_postfix** descriptor that defines the postfix for the **cmd_to_discover** descriptor in an **sm_cmd_hdr** object defining a dialog header.
- The **cmd_to_classify_postfix** descriptor that defines the postfix for the **cmd_to_classify** descriptor in an **sm_name_hdr** object defining a selector header.
- The **cmd_to_list_postfix** descriptor that defines the postfix for the **cmd_to_list** descriptor in an **sm_cmd_opt** object defining a selector entry field associated with a selector or a dialog entry field associated with a dialog.

The following is an example of how the postfix descriptors are used to specify parameter flags and values. The * (asterisk) in the example can be `list`, `classify`, or `discover`.

Assume that `cmd_to_*` equals `"DEMO -a"`, that `cmd_to_*_postfix` equals `"-l _rawname -n stuff -R _cookedname"`, and that the current value set is:

```
#name1:_rawname:_cookedname::stuff\n
value1:gigatronicundulator:parallel:xxx:47
```

Then the constructed command string would be:

```
DEMO -a -l 'gigatronicundulator' -n '47' -R 'parallel'
```

Surrounding '' (single-quotation marks) can be added around postfix descriptor values to permit handling of parameter values with embedded spaces.

SMIT Command Generation and Execution

Each dialog in the System Management Interface Tool (SMIT) builds and executes a version of a standard command. The command to be executed by the dialog is defined by the **cmd_to_exec** descriptor in the **sm_cmd_hdr** object that defines the dialog header.

Generating Dialog Defined Tasks

In building the command defined in an **sm_cmd_hdr.cmd_to_exec** descriptor, SMIT uses a two-pass scan over the dialog set of **sm_cmd_opt** objects to collect prefix and parameter values. The parameter values collected include those that the user changed from their initially displayed values and those with the **sm_cmd_opt.required** descriptor set to `"y"`.

The first pass gathers all of the values of the **sm_cmd_opt** objects (in order) for which the **prefix** descriptor is either an empty string (`""`) or starts with a - (a minus sign). These parameters are not position-sensitive and are added immediately following the command name, together with the contents of the **prefix** descriptor for the parameter.

The second pass gathers all of the values of the remaining **sm_cmd_opt** objects (in order) for which the **prefix** descriptor is – (two dashes). These parameters are position-sensitive and are added after the flagged options collected in the first pass.

> **Note:** SMIT executes the value of what you enter in the prefix field. If the value in the prefix field is a reserved shell character, for example, the * (asterisk), you must follow the character with a —' (dash dash single quotation mark). Then, when the system evaluates the character, it does not mistake it for a shell character.

Command parameter values in a dialog are filled in automatically when the **disc_field_name** descriptors of its **sm_cmd_opt** objects match names of values generated by preceding selectors or a preceding discovery command. These parameter values are effectively default values and are normally not added to the command line. Initializing an **sm_cmd_opt.required** descriptor to `"y"` or `"+"` causes these values to be added to the command line even when they are not changed in the dialog. If the **sm_cmd_opt.required** descriptor value is `"?"`, the corresponding values are used only if the associated entry field is non-empty. These parameter values are built into the command line as part of the regular two-pass process.

Leading and trailing white spaces (spaces and tabs) are removed from parameter values except when the **sm_cmd_opt.entry_type** descriptor is set to `"r"`. If the resulting parameter value is an empty string, no further action is taken unless the **sm_cmd_opt.prefix** descriptor starts with an option flag. Surrounding single quotation marks are added to the parameter value if the **prefix** descriptor is not set to `"–"` (two

dashes). Each parameter is placed immediately after the associated prefix, if any, with no intervening spaces. Also, if the **multi_select** descriptor is set to ″m″, tokens separated by white space in the entry field are treated as separate parameters.

Executing Dialog Defined Tasks

SMIT runs the command string specified in a **sm_cmd_hdr.cmd_to_exec** descriptor by first creating a child process. The standard error and standard output of the child process are handled as specified by the contents of the **sm_cmd_hdr.exec_mode** descriptor. SMIT next runs a **setenv(″ENV=″)** subroutine in the child process to prevent commands specified in the **$HOME/.env** file of the user from being run automatically when a new shell is invoked. Finally, SMIT calls the **execl** subroutine to start a **ksh** shell, using the command string as the **ksh -c** parameter value.

SMIT makes the path names of the log files and the settings of the command line verbose, trace, and debug flags available in the shell environment of the commands it runs. These values are provided with the following environment variables:

- **_SMIT_LOG_FILE**
- **_SMIT_SCRIPT_FILE**
- **_SMIT_VERBOSE_FLAG**
- **_SMIT_TRACE_FLAG**
- **_SMIT_DEBUG_FLAG**

The presence or absence of the corresponding flag is indicated by a value of 0 or 1, respectively.

Additionally, the **SMIT** environment variable provides information about which SMIT environment is active. The **SMIT** environment variable can have the following vaues:

Value	SMIT Environment
a	SMIT in an ASCII interface
d	SMIT in the Distributed SMIT (DSMIT) interface
m	SMIT in a windows (also called Motif) interface

An easy way to view the current settings is to invoke the shell function after starting SMIT and then run the command string **env | grep SMIT**.

You can disable the function key F9=Shell by setting the environment variable **SMIT_SHELL=n**.

All writes to the log files should be done as appends and should immediately be followed by flushes where this does not occur automatically.

You can override SMIT default output redirection of the (child) task process by setting the `sm_cmd_hdr.exec_mode` field to ″i″. This setting gives output management control to the task, since the task process simply inherits the standard error and standard output file descriptors.

You can cause SMIT to shutdown and replace itself with the target task by setting the `sm_cmd_hdr.exec_mode` field to ″e″.

Adding Tasks to the SMIT Database

When developing new objects for the System Management Interface Tool (SMIT) database, it is recommended that you set up a separate test database for development.

Procedure

To create a test database, do the following:

1. Create a directory for testing use. For example, the following command creates a `/home/smit/test` directory:

   ```
   mkdir /home/smit /home/smit/test
   ```

2. Make the test directory the current directory:

   ```
   cd /home/smit/test
   ```

3. Define the test directory as the default object repository by setting the **ODMDIR** environment variable to **.** (the current directory):

   ```
   export ODMDIR=
   ```

4. Create a new SMIT database in the test directory:

   ```
   cp /etc/objrepos/sm_* $ODMDIR
   ```

To add tasks to the SMIT database:

1. Design the dialog for the command you want SMIT to build. See “Dialog Screens” on page 661 for more information.
2. Design the hierarchy of menus and, optionally, of selectors needed to get a SMIT user to the dialog, and determine where and how this hierarchy should be linked into the existing SMIT database. See “Menu Screens” on page 659 and “Selector Screens” on page 660 for more information. The following strategy may save you time if you are developing SMIT database extensions for the first time:
   a. Start SMIT (run the **smit** command), look for existing menu, selector, and dialog screens that perform tasks similar to the one you want to add, and find the menu screen(s) to which you will add the new task.
   b. Exit from SMIT, then remove the existing SMIT log file. Instead of removing the log file, you can use the **-l** flag of the **smit** command to specify a different log file when starting SMIT in the following step. This enables you to isolate the trace output of your next SMIT session.
   c. Start SMIT again with the **-t** command flags and again look at the screen to which you will add the new task. This logs the object IDs accessed for each screen for the next step.
   d. Look at the SMIT log file to determine the ID for each object class used as part of the menu(s).
   e. Use the object class IDs with the **odmget** command to retrieve the stanzas for these objects. The stanzas can be used as rough examples to guide your implementation and to learn from the experience of others.
   f. Look in the SMIT log file for the command strings used when running through the screens to see if special tools are being utilized (such as **sed** or **awk** scripts, **ksh** shell functions, environment variable assignment, and so on). When entering command strings, keep in mind that they are processed twice: the first time by the **odmadd** command and the second time by the **ksh** shell. Be careful when using special escape meta-characters such as \ or quotation characters (’ and ″). Note also that the output of the **odmget** command does not always match the input to the **odmadd** command, especially when these characters or multiline string values are used.
3. Code the dialog, menu, and selector objects by defining them in the ASCII object stanza file format required by the **odmadd** command. For examples of stanzas used to code SMIT objects, see “SMIT Screen Types” on page 659.
4. Add the dialog, menu, and selector objects to the SMIT test database with the **odmadd** command, using the name of your ASCII object stanza file in place of `test_stanzas`:

   ```
   odmadd test_stanzas
   ```

5. Test and debug your additions by running SMIT using the local test database:

   ```
   smit -o
   ```

   “Debugging SMIT Database Extensions” on page 671 discusses how to test and debug additions to SMIT.

When you are finished testing, restore the **/usr/lib/objrepos** directory as the default object repository by setting the **ODMDIR** environment variable to **/usr/lib/objrepos**:

```
export ODMDIR=/usr/lib/objrepos
```

Debugging SMIT Database Extensions

Prerequisite Tasks or Conditions

1. Add a task to the SMIT Database.
2. Test the task.

Procedure

1. Identify the problem using one of the following flags:
   - Run the **smit -v** command if the problem applies to the following SMIT descriptors:
     - **cmd_to_list**
     - **cmd_to_classify**
     - **cmd_to_discover**
   - Run the **smit -t** command if the problems applies to individual SMIT database records.
   - Run the **smit -l** command to generate an alternate log file. Use the alternate log file to isolate current session information.
2. Modify the SMIT database where the incorrect information resides.
3. Retest SMIT task.

Creating SMIT Help Information for a New Task

System Management Interface Tool (SMIT) helps are an extension of the SMIT program. They are a series of helps designed to give you online information about the components of SMIT used to construct dialogs and menus. SMIT helps reside in a database, just as the SMIT executable code resides in a database. SMIT has three ways to retrieve SMIT help information:

- "Man Pages Method"
- "Message Catalog Method" on page 672
- "Softcopy Libraries Method" on page 672.

Each of these methods provides a different way to retrieve SMIT helps from the SMIT help database.

Man Pages Method

Prerequisite Tasks or Conditions

Create a new SMIT task that requires help information.

Procedure

1. Using any editor, create a file and enter help text inside the file. The file must adhere to the format specified by the **man** command. Put only one set of help information in a file.
2. Give the help text file a title as specified by the **man** command.
3. Place the help text file in the correct place in the **manual** subdirectory.
4. Test the newly created file to ensure it works using the **man** command.
5. Locate the file that contains the ASCII object stanza file format for the new SMIT task.
6. Locate the help descriptor fields in the object stanzas of the file.
7. Set the `help_msg_loc` help descriptor field equal to the title of the help text file. The title for the text file is also the parameter to pass to the **man** command. For example:

```
help_msg_loc = "xx", where "xx" = title string name
```

This example executes the **man** command with the xx title string name.

8. Leave the rest of the help descriptor fields empty.

Message Catalog Method

Prerequisite Tasks or Conditions

Create a new SMIT task that requires help information.

Procedure

1. Use any editor to create a file and enter help messages inside the file. The **.msg** file must adhere to the format specified by the “Message Facility Overview for Programming” on page 480.

   Note: An existing **.msg** file may also be used.
2. Give each help message a set number (Set #) and a message number (MSG#). This allows the system to retrieve the proper help text.
3. Use the **gencat** command to convert the **.msg** file into a **.cat** file. Place the **.cat** file in the correct directory according to the **NLSPATH** environment variable.
4. Test the help messages using the **dspmsg** command.
5. Locate the file that contains the ASCII object stanza file format for the new SMIT task.
6. Locate the help descriptor fields in the object stanzas of the file.
7. For each object stanza, locate the help_msg_id help descriptor field. Enter the Set# and Msg# values for the message in the **.msg** file. These values must adhere to the Messages Facility format. For example, to retrieve message #14 for set #2, set:

   ```
   help_msg_id - "2,14"
   ```
8. Set the help_msg_loc help descriptor field to the filename of the file containing the help text.
9. Leave the other help descriptor fields empty.

Softcopy Libraries Method

Prerequisite Tasks or Conditions

Create a new SMIT task that requires help information.

InfoCrafter must be installed on your system in order to create softcopy files.

Procedure

1. Create a softcopy database file that tags the SMIT help information with the proper SMiT ID. SMiT IDs are hidden search strings that are given to text in a softcopy database. SMiT tags take the format of SMiTtopic#tag#, where topic#tag# is a numeric string of at least 4 digits and up to 8 digits. For example:

   ```
   SMiT0822369 is a SMiT ID tag for TCP/IP
   ```
2. Create a softcopy library file to create a pointer to the softcopy database file. For details on the format of the library file, refer to the **ispaths** file.
3. Set the help_msg_id help descriptor field to equal the numeric string of the SMiT ID tag. For example, if the SMiT identifier tag is SMiT0822369, then set:

   ```
   help_msg_id = "0822369"
   ```
4. Set the help_msg_base help descriptor field to equal the full path name of the **ispaths** file created in step 2. SMIT reads the file for the softcopy database associated with the correct book.
5. Set the help_msg_book help descriptor field to equal the value of the name field contained in the file indicated by the help_msg_base help descriptor field.

sm_menu_opt (SMIT Menu) Object Class

Each item on a menu is specified by an **sm_menu_opt** object. The displayed menu represents the set of objects that have the same value for **id** plus the **sm_menu_opt** object used for the title, which has a **next_id** value equal to the **id** value of the other objects.

> **Note:** When coding an object in this object class, set unused empty strings to "" (double-quotation marks) and unused integer fields to 0.

The descriptors for **sm_menu_opt** objects are:

id The ID or name of the object. The value of **id** is a string with a maximum length of 64 characters. IDs should be unique both to your application and unique within the particular SMIT database used. See the **next_id** and **alias** definitions for this object for related information.

id_seq_num The position of this item in relation to other items on the menu. Non-title **sm_menu_opt** objects are sorted on this string field. The value of **id_seq_num** is a string with a maximum length of 16 characters.

next_id The fast path name of the next menu, if the value for the **next_type** descriptor of this object is "m" (menu). The **next_id** of a menu should be unique both to your application and within the particular SMIT database used. All non-alias **sm_menu_opt** objects with **id** values matching the value of **next_id** form the set of selections for that menu. The value of **next_id** is a string with a maximum length of 64 characters.

text The description of the task that is displayed as the menu item. The value of **text** is a string with a maximum length of 1024 characters. This string can be formatted with embedded \n (newline) characters.

text_msg_file The file name (not the full path name) that is the Message Facility catalog for the string, **text**. The value of **text_msg_file** is a string with a maximum length of 1024 characters. Message catalogs required by an application program can be developed with the Message Facility. Set to "" if you are not using the Message Facility.

text_msg_set The Message Facility set ID for the string, **text**. Set IDs can be used to indicate subsets of a single catalog. The value of **text_msg_set** is an integer. Set to 0 if you are not using the Message Facility.

text_msg_id The Message Facility ID for the string, **text**. The value of **text_msg_id** is an integer. Set to 0 if you are not using the Message Facility.

next_type The type of the next object if this item is selected. Valid values are:

- "**m**" Menu; the next object is a menu (**sm_menu_opt**).
- "**d**" Dialog; the next object is a dialog (**sm_cmd_hdr**).
- "**n**" Name; the next object is a selector (**sm_name_hdr**).
- "**i**" Info; this object is used to put blank or other separator lines in a menu, or to present a topic that does not lead to an executable task but about which help is provided via the **help_msg_loc** descriptor of this object.

alias Defines whether or not the value of the **id** descriptor for this menu object is an alias for another existing fast path specified in the next_id field of this object. The value of the **alias** descriptor must be "n" for a menu object.

help_msg_id Specifies a Message Facility message set number and message ID number with a comma as the separator or a numeric string equal to a SMIT identifier tag.

help_msg_loc The file name sent as a parameter to the **man** command for retrieval of help text, or the file name of a file containing help text. The value of **help_msg_loc** is a string with a maximum length of 1024 characters.

help_msg_base The fully qualified path name of a library that SMIT reads for the file names associated with the correct book.

help_msg_book Contains the string with the value of the name file contained in the file library indicated by **help_msg_base**.

The sm_menu_opt Object Class Used for Aliases

A SMIT alias is specified by an **sm_menu_opt** object.

The descriptors for the **sm_menu_opt** object class and their settings to specify an alias are:

id	The ID or name of the new or alias fast path. The value of **id** is a string with a maximum length of 64 characters. IDs should be unique to your application and unique within the SMIT database in which they are used.
id_seq_num	Set to "" (empty string).
next_id	Specifies the **id_seq_num** of the menu object pointed to by the alias. The value of **next_id** is a string with a maximum length of 64 characters.
text	Set to "" (empty string).
text_msg_file	Set to "" (empty string).
text_msg_set	Set to 0.
text_msg_id	Set to 0.
next_type	The fast path screen type. The value of **next_type** is a string. Valid values are: "**m**" Menu; the next_id field specifies a menu screen fast path. "**d**" Dialog; the next_id field specifies a dialog screen fast path. "**n**" Name; the next_id field specifies a selector screen fast path.
alias	Defines this object as an alias fast path. The **alias** descriptor for an alias must be set to "y" (yes).
help_msg_id	Set to "" (empty string).
help_msg_loc	Set to "" (empty string).
help_msg_base	Set to "" (empty string).
help_msg_book	Set to "" (empty string).

For information on retrieving SMIT help using the help_msg_id, help_msg_loc, help_msg_base, and help_msg_book fields, see the "Man Pages Method" on page 671, "Softcopy Libraries Method" on page 672, and "Message Catalog Method" on page 672 methods located in "Creating SMIT Help Information for a New Task .

sm_name_hdr (SMIT Selector Header) Object Class

A selector screen is specified by two objects: an **sm_name_hdr** object that specifies the screen title and other information, and an **sm_cmd_opt** object that specifies what type of data item is to be obtained.

> **Note:** When coding an object in this object class, set unused empty strings to "" (double-quotation marks) and unused integer fields to 0.

In a SMIT Selector Header screen (**sm_name_hdr**) with type = "c", if you specify a value using a : (colon), (for example, tty:0), SMIT inserts a #! (pound sign, exclamation point) in front of the : to signify that the : is not a field separator. SMIT removes the #! after parsing the rest of the value, before passing it to the **cmd_to_classify** descriptor. To make any further additions to the **cmd_to_classify** descriptor, reinsert the #! in front of the :

The descriptors for the **sm_name_hdr** object class are:

id	The ID or name of the object. The id field can be externalized as a fast path ID unless **has_name_select** is set to "y" (yes). The value of **id** is a string with a maximum length of 64 characters. IDs should be unique to your application and unique within your system.

next_id
Specifies the header object for the subsequent screen; set to the value of the `id` field of the **sm_cmd_hdr** object or the **sm_name_hdr** object that follows this selector. The `next_type` field described below specifies which object class is indicated. The value of **next_id** is a string with a maximum length of 64 characters.

option_id
Specifies the body of this selector; set to the `id` field of the **sm_cmd_opt** object. The value of **option_id** is a string with a maximum length of 64 characters.

has_name_select
Specifies whether this screen must be preceded by a selector screen. Valid values are:

″″ **or** ″**n**″
: No; this is the default case. The **id** of this object can be used as a fast path, even if preceded by a selector screen.

″**y**″
: Yes; a selector must precede this object. This setting prevents the **id** of this object from being used as a fast path to the corresponding dialog screen.

name
The text displayed as the title of the selector screen. The value of **name** is a string with a maximum length of 1024 characters. The string can be formatted with embedded `\n` (newline) characters.

name_msg_file
The file name (not the full path name) that is the Message Facility catalog for the string, **name**. The value of **name_msg_file** is a string with a maximum length of 1024 characters. Message catalogs required by an application program can be developed with the Message Facility.

name_msg_set
The Message Facility set ID for the string, **name**. Set IDs can be used to indicate subsets of a single catalog. The value of **name_msg_set** is an integer.

name_msg_id
The Message Facility ID for the string, **name**. The value of **name_msg_id** is an integer.

type
The method to be used to process the selector. The value of **type** is a string with a maximum length of 1 character. Valid values are:

″″ **or** ″**j**″
: Just next ID; the object following this object is always the object specified by the value of the **next_id** descriptor. The **next_id** descriptor is a fully-defined string initialized at development time.

″**r**″
: Cat raw name; in this case, the **next_id** descriptor is defined partially at development time and partially at runtime by user input. The value of the **next_id** descriptor defined at development time is concatenated with the value selected by the user to create the **id** value to search for next (that of the dialog or selector to display).

″**c**″
: Cat cooked name; the value selected by the user requires processing for more information. This value is passed to the command named in the **cmd_to_classify** descriptor, and then output from the command is concatenated with the value of the **next_id** descriptor to create the **id** descriptor to search for next (that of the dialog or selector to display).

ghost
Specifies whether to display this selector screen or only the list pop-up panel produced by the command in the `cmd_to_list` field. The value of **ghost** is a string. Valid values are:

″″ **or** ″**n**″
: No; display this selector screen.

″**y**″
: Yes; display only the pop-up panel produced by the command string constructed using the `cmd_to_list` and `cmd_to_list_postfix` fields in the associated **sm_cmd_opt** object. If there is no **cmd_to_list** value, SMIT assumes this object is a super-ghost (nothing is displayed), runs the **cmd_to_classify** command, and proceeds.

cmd_to_classify
The command string to be used, if needed, to classify the value of the `name` field of the **sm_cmd_opt** object associated with this selector. The value of **cmd_to_classify** is a string with a maximum length of 1024 characters. The input to the **cmd_to_classify** taken from the `entry` field is called the ″raw name″ and the output of the **cmd_to_classify** is called the ″cooked name″. Previous to AIX Version 4.2.1, you could create only one value with **cmd_to_classify**. If that value included a colon, it was escaped automatically. In AIX 4.2.1 and later, you can create multiple values with **cmd_to_classify**, but the colons are no longer escaped. The colon is now being used as a delimiter by this command. If you use colons in your values, you must preserve them manually.

cmd_to_classify_postfix
The postfix to interpret and add to the command string in the `cmd_to_classify` field. The value of **cmd_to_classify_postfix** is a string with a maximum length of 1024 characters.

raw_field_name
The alternate name for the raw value. The value of **raw_field_name** is a string with a maximum length of 1024 characters. The default value is ″`_rawname`″.

cooked_field_name
The alternate name for the cooked value. The value of **cooked_field_name** is a string with a maximum length of 1024 characters. The default value is ″`cookedname`″.

next_type
The type of screen that follows this selector. Valid values are:

″**n**″
: Name; a selector screen follows. See the description of **next_id** above for related information.

″**d**″
: Dialog; a dialog screen follows. See the description of **next_id** above for related information.

help_msg_id
Specifies a Message Facility message set number and message ID number with a comma as the separator or a numeric string equal to a SMIT identifier tag.

help_msg_loc
The file name sent as a parameter to the **man** command for retrieval of help text, or the file name of a file containing help text. The value of **help_msg_loc** is a string with a maximum length of 1024 characters.

help_msg_base
The fully qualified path name of a library that SMIT reads for the file names associated with the correct book.

help_msg_book
Contains the string with the value of the name file contained in the file library indicated by **help_msg_base**.

See the “Man Pages Method” on page 671, “Softcopy Libraries Method” on page 672, and “Message Catalog Method” on page 672 located in ″Creating SMIT Help Information for a New Task″ for information on retrieving SMIT help using the `help_msg_id`, `help_msg_loc`, `help_msg_base`, and `help_msg_book` fields.

sm_cmd_opt (SMIT Dialog/Selector Command Option) Object Class

Each object in a dialog, except the dialog header object, normally corresponds to a flag, option, or attribute of the command that the dialog performs. One or more of these objects is created for each SMIT dialog; a ghost dialog can have no associated dialog command option objects. Each selector screen is composed of one selector header object and one selector command option object.

> **Note:** When coding an object in this object class, set unused empty strings to "" (double-quotation marks) and unused integer fields to 0.

The dialog command option object and the selector command option object are both **sm_cmd_opt** objects. The descriptors for the **sm_cmd_opt** object class and their functions are:

id
The ID or name of the object. The **id** of the associated dialog or selector header object can be used as a fast path to this and other dialog objects in the dialog. The value of **id** is a string with a maximum length of 64 characters. All dialog objects that appear in one dialog must have the same ID. Also, IDs should be unique to your application and unique within the particular SMIT database used.

id_seq_num
The position of this item in relation to other items on the dialog; **sm_cmd_opt** objects in a dialog are sorted on this string field. The value of **id_seq_num** is a string with a maximum length of 16 characters. When this object is part of a dialog screen, the string "0" is not a valid value for this field. When this object is part of a selector screen, the **id_seq_num** descriptor must be set to 0.

disc_field_name
A string that should match one of the name fields in the output of the **cmd_to_discover** command in the associated dialog header. The value of **disc_field_name** is a string with a maximum length of 64 characters.

The value of the **disc_field_name** descriptor can be defined using the raw or cooked name from a preceding selector instead of the **cmd_to_discover** command in the associated header object. If the descriptor is defined with input from a preceding selector, it must be set to either "_rawname" or "_cookedname", or to the corresponding sm_name_hdr.cooked_field_name value or sm_name_hdr.raw_field_name value if this was used to redefine the default name.

name
The string that appears on the dialog or selector screen as the field name. It is the visual questioning or prompting part of the object, a natural language description of a flag, option or parameter of the command specified in the cmd_to_exec field of the associated dialog header object. The value of **name** is a string with a maximum length of 1024 characters.

name_msg_file
The file name (not the full path name) that is the Message Facility catalog for the string, **name**. The value of **name_msg_file** is a string with a maximum length of 1024 characters. Message catalogs required by an application program can be developed with the Message Facility. Set to "" (empty string) if not used.

name_msg_set
The Message Facility set ID for the string, **name**. The value of **name_msg_set** is an integer. Set to 0 if not used.

name_msg_id
The Message Facility message ID for the string, **name**. The value of **name_msg_id** is an integer. Set to 0 if not used.

op_type
The type of auxiliary operation supported for this field. The value of **op_type** is a string. Valid values are:

"" or "**n**" - This is the default case. No auxiliary operations (list or ring selection) are supported for this field.

"**l**" - List selection operation provided. A pop-up window displays a list of items produced by running the command in the `cmd_to_list` field of this object when the user selects the **F4=List** function of the SMIT interface.

"**r**" - Ring selection operation provided. The string in the `disp_values` or `aix_values` field is interpreted as a comma-delimited set of valid entries. The user can tab or backtab through these values to make a selection. Also, the **F4=List** interface function can be used in this case, since SMIT will transform the ring into a list as needed.

The values "`N`", "`L`", and "`R`" can be used as **op_type** values just as the lowercase values "`n`", "`l`", and "`r`". However, with the uppercase values, if the **cmd_to_exec** command is run and returns with an exit value of `0`, then the corresponding entry field will be cleared to an empty string.

entry_type
The type of value required by the entry field. The value of **entry_type** is a string. Valid values are:

"" or "**n**" - No entry; the current value cannot be modified via direct type-in. The field is informational only.

"**t**" - Text entry; alphanumeric input can be entered.

"**#**" - Numeric entry; only the numeric characters `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, or `9` can be entered. A - (minus sign) or + (plus sign) can be entered as the first character.

"**x**" - Hex entry; hexadecimal input only can be entered.

"**f**" - File entry; a file name should be entered.

"**r**" - Raw text entry; alphanumeric input can be entered. Leading and trailing spaces are considered significant and are not stripped off the field.

entry_size
Limits the number of characters the user can type in the entry field. The value of **entry_size** is an integer. A value of `0` defaults to the maximum allowed value size.

required
Defines if a command field must be sent to the **cmd_to_exec** command defined in the associated dialog header object. The value of **required** is a string. If the object is part of a selector screen, the `required` field should normally be set to "" (empty string). If the object is part of a dialog screen, valid values are:

"" or "**n**" - No; the option is added to the command string in the **cmd_to_exec** command only if the user changes the initially-displayed value. This is the default case.

"**y**" - Yes; the value of the `prefix` field and the value of the entry field are always sent to the **cmd_to_exec** command.

"**+**" - The value of the `prefix` field and the value of the entry field are always sent to the **cmd_to_exec** command. The entry field must contain at least one non-blank character. SMIT will not allow the user to run the task until this condition is satisfied.

"**?**" - Except when empty; the value of the `prefix` field and the value of the entry field are sent to the `cmd_to_exec` field unless the entry field is empty.

prefix
In the simplest case, defines the flag to send with the entry field value to the **cmd_to_exec** command defined in the associated dialog header object. The value of **prefix** is a string with a maximum length of 1024 characters.

The use of this field depends on the setting of the `required` field, the contents of the `prefix` field, and the contents of the associated `entry` field.

> **Note:** If the `prefix` field is set to – (dash dash), the content of the associated entry field is appended to the end of the **cmd_to_exec** command. If the `prefix` field is set to –' (dash dash single quotation mark), the contents of the associated entry field is appended to the end of the **cmd_to_exec** command in single quotes.

cmd_to_list_mode
Defines how much of an item from a list should be used. The list is produced by the command specified in this object's `cmd_to_list` field. The value of **cmd_to_list_mode** is a string with a maximum length of 1 character. Valid values are:

″″ or ″**a**″ - Get all fields. This is the default case.

″**1**″ - Get the first field.

″**2**″ - Get the second field.

″**r**″ - Range; running the command string in the `cmd_to_list` field returns a range (such as **1..99**) instead of a list. Ranges are for information only; they are displayed in a list pop-up, but do not change the associated entry field.

cmd_to_list
The command string used to get a list of valid values for the value field. The value of **cmd_to_list** is a string with a maximum length of 1024 characters. This command should output values that are separated by `\n` (new line) characters.

cmd_to_list_postfix
The postfix to interpret and add to the command string specified in the `cmd_to_list` field of the dialog object. The value of **cmd_to_list_postfix** is a string with a maximum length of 1024 characters. If the first line starts with `#` (pound sign) following a space, that entry will be made non-selectable. This is useful for column headings. Subsequent lines that start with a `#`, optionally preceded by spaces, are treated as a comment and as a continuation of the preceding entry.

multi_select
Defines if the user can make multiple selections from a list of valid values produced by the command in the `cmd_to_list` field of the dialog object. The value of **multi_select** is a string. Valid values are:

″″ - No; a user can select only one value from a list. This is the default case.

″**,**″ - Yes; a user can select multiple items from the list. When the command is built, a comma is inserted between each item.

″**y**″ - Yes; a user can select multiple values from the list. When the command is built, the option prefix is inserted once before the string of selected items.

″**m**″ - Yes; a user can select multiple items from the list. When the command is built, the option prefix is inserted before each selected item.

value_index
For an option ring, the zero-origin index to the array of `disp_value` fields. The **value_index** number indicates the value that is displayed as the default in the entry field to the user. The value of **entry_size** is an integer.

disp_values
The array of valid values in an option ring to be presented to the user. The value of the `disp_values` fields is a string with a maximum length of 1024 characters. The field values are separated by **,** (commas) with no spaces preceding or following the commas.

values_msg_file The file name (not the full path name) that is the Message Facility catalog for the values in the `disp_values` fields, if the values are initialized at development time. The value of the `values_msg_file` field is a string with a maximum length of 1024 characters. Message catalogs required by an application program can be developed with the Message Facility.

values_msg_set The Message Facility set ID for the values in the `disp_values` fields. Set to 0 if not used.

values_msg_id The Message Facility message ID for the values in the `disp_values` fields. Set to 0 if not used.

aix_values If for an option ring, an array of values specified so that each element corresponds to the element in the **disp_values** array in the same position; use if the natural language values in **disp_values** are not the actual options to be used for the command. The value of the `aix_values` field is a string with a maximum length of 1024 characters.

help_msg_id Specifies a Message Facility message set number and message ID number with a comma as the separator or a numeric string equal to a SMIT identifier tag.

help_msg_loc The file name sent as a parameter to the **man** command for retrieval of help text, or the file name of a file containing help text. The value of **help_msg_loc** is a string with a maximum length of 1024 characters.

help_msg_base The fully qualified path name of a library that SMIT reads for the file names associated with the correct book.

help_msg_book Contains the string with the value of the name file contained in the file library indicated by **help_msg_base**.

For information on retrieving SMIT help using the `help_msg_id`, `help_msg_loc`, `help_msg_base`, and `help_msg_book` fields, see “Man Pages Method” on page 671, “Softcopy Libraries Method” on page 672, and “Message Catalog Method” on page 672 located in ″“Creating SMIT Help Information for a New Task” on page 671″ .

sm_cmd_hdr (SMIT Dialog Header) Object Class

A dialog header object is an **sm_cmd_hdr** object. A dialog header object is required for each dialog, and points to the dialog command option objects associated with the dialog.

> **Note:** When coding an object in this object class, set unused empty strings to ″″ (double-quotation marks) and unused integer fields to 0.

The descriptors for the **sm_cmd_hdr** object class are:

id The ID or name of the object. The value of **id** is a string with a maximum length of 64 characters. The `id` field can be used as a fast path ID unless there is a selector associated with the dialog. IDs should be unique to your application and unique within your system.

option_id The **id** of the **sm_cmd_opt** objects (the dialog fields) to which this header refers. The value of **option_id** is a string with a maximum length of 64 characters.

has_name_select Specifies whether this screen must be preceded by a selector screen or a menu screen. Valid values are:

″″ **or** ″**n**″
: No; this is the default case.

″**y**″
: Yes; a selector precedes this object. This setting prevents the **id** of this object from being used as a fast path to the corresponding dialog screen.

name
The text displayed as the title of the dialog screen. The value of **name** is a string with a maximum length of 1024 characters. The text describes the task performed by the dialog. The string can be formatted with embedded **\n** (newline) characters.

name_msg_file
The file name (not the full path name) that is the Message Facility catalog for the string, **name**. The value of **name_msg_file** is a string with a maximum length of 1024 characters. Message catalogs required by an application program can be developed with the Message Facility.

name_msg_set
The Message Facility set ID for the string, **name**. Set IDs can be used to indicate subsets of a single catalog. The value of **name_msg_set** is an integer.

name_msg_id
The Message Facility ID for the string, **name**. Message IDs can be created by the message extractor tools owned by the Message Facility. The value of **name_msg_id** is an integer.

cmd_to_exec
The initial part of the command string, which can be the command or the command and any fixed options that execute the task of the dialog. Other options are automatically appended through user interaction with the command option objects (**sm_cmd_opt**) associated with the dialog screen. The value of **cmd_to_exec** is a string with a maximum length of 1024 characters.

ask
Defines whether or not the user is prompted to reconsider the choice to execute the task. Valid values are:

″″ **or** ″**n**″
: No; the user is not prompted for confirmation; the task is performed when the dialog is committed. This is the default setting for the **ask** descriptor.

″**y**″
: Yes; the user is prompted to confirm that the task be performed; the task is performed only after user confirmation.

  Prompting the user for execution confirmation is especially useful prior to performance of deletion tasks, where the deleted resource is either difficult or impossible to recover, or when there is no displayable dialog associated with the task (when the `ghost` field is set to ″`y`″).

exec_mode

Defines the handling of standard input, standard output, and the **stderr** file during task execution. The value of **exec_mode** is a string. Valid values are:

″″ **or** ″**p**″
: Pipe mode; the default setting for the **exec_mode** descriptor. The command executes with standard output and the **stderr** file redirected through pipes to SMIT. SMIT manages output from the command. The output is saved and is scrollable by the user after the task finishes running. While the task runs, output is scrolled as needed.

″**n**″
: No scroll pipe mode; works like the ″p″ mode, except that the output is not scrolled while the task runs. The first screen of output will be shown as it is generated and then remains there while the task runs. The output is saved and is scrollable by the user after the task finishes running.

″**i**″
: Inherit mode; the command executes with standard input, standard output, and the **stderr** file inherited by the child process in which the task runs. This mode gives input and output control to the executed command. This value is intended for commands that need to write to the **/dev/tty** file, perform cursor addressing, or use **libcur** or **libcurses** library operations.

″**e**″
: Exit/exec mode; causes SMIT to run (do an **execl** subroutine call on) the specified command string in the current process, which effectively terminates SMIT. This is intended for running commands that are incompatible with SMIT (which change display modes or font sizes, for instance). A warning is given that SMIT will exit before running the command.

″**E**″
: Same as ″**e**″, but no warning is given before exiting SMIT.

″**P**″ **,** ″**N**″ **or** ″**I**″
: Backup modes; work like the corresponding ″p″, ″n″, and ″i″ modes, except that if the **cmd_to_exec** command is run and returns with an exit value of 0, SMIT backs up to the nearest preceding menu (if any), or to the nearest preceding selector (if any), or to the command line.

ghost

Indicates if the normally displayed dialog should not be shown. The value of **ghost** is a string. Valid values are:

″″ **or** ″**n**″
: No; the dialog associated with the task is displayed. This is the default setting.

″**y**″
: Yes; the dialog associated with the task is not displayed because no further information is required from the user. The command specified in the **cmd_to_exec** descriptor is executed as soon as the user selects the task.

cmd_to_discover

The command string used to discover the default or current values of the object being manipulated. The value of **cmd_to_discover** is a string with a maximum length of 1024 characters. The command is executed before the dialog is displayed, and its output is retrieved. Output of the command must be in colon format.

cmd_to_discover_postfix

The postfix to interpret and add to the command string in the `cmd_to_discover` field. The value of **cmd_to_discover_postfix** is a string with a maximum length of 1024 characters.

help_msg_id	Specifies a Message Facility message set number and message ID number with a comma as the separator or a numeric string equal to a SMIT identifier tag.
help_msg_loc	The file name sent as a parameter to the **man** command for retrieval of help text, or the file name of a file containing help text. The value of **help_msg_loc** is a string with a maximum length of 1024 characters.
help_msg_base	The fully qualified path name of a library that SMIT reads for the file names associated with the correct book.
help_msg_book	Contains the string with the value of the name file contained in the file library indicated by **help_msg_base**.

For information on retrieving SMIT help using the `help_msg_id`, `help_msg_loc`, `help_msg_base`, and `help_msg_book` fields, see “Man Pages Method” on page 671, “Softcopy Libraries Method” on page 672, and “Message Catalog Method” on page 672 located in “Creating SMIT Help Information for a New Task” on page 671.

Related Information

For information about managing SMIT, see System Management Interface Tool (SMIT): Overview in *AIX 5L Version 5.1 System Management Guide: Operating System and Devices*.

The **dspmsg** command, **gencat** command, **ksh** command, **man** command, **odmadd** command, **odmcreate** command, **odmget** command, **smit** command in *AIX 5L Version 5.1 Commands Reference*.

The **ispaths** file in *AIX 5L Version 5.1 Files Reference*.

SMIT Example Program

The following example program is designed to help you write your own stanzas. If you add these stanzas to the SMIT directory that comes with the operating system, they will be accessible through SMIT by selecting the **Applications** item in the SMIT main menu. All of the demos are functional except for Demo 3, which does not install any languages.

```
#------------------------------------------------------------------
# Intro:
# Unless you are creating a new SMIT database, first you need
# to decide where to insert the menu for your application.
# Your new menu will point to other menus, name headers, and
# dialogs. For this example, we are inserting a pointer to the
# demo menu under the "Applications" menu option. The next_id for
# the Applications menu item is "apps", so we begin by creating a
# menu_opt with "apps" as its id.
#------------------------------------------------------------------
sm_menu_opt:
   id           = "apps"
   id_seq_num   = "010"
   next_id      = "demo"
   text         = "SMIT Demos"
   next_type    = "m"

sm_menu_opt:
   id           = "demo"
   id_seq_num   = "010"
   next_id      = "demo_queue"
   text         = "Demo 1: Add a Print Queue"
   next_type    = "n"

sm_menu_opt:
   id              = "demo"
   id_seq_num      = "020"
   next_id         = "demo_mle_inst_lang_hdr"
```

```
   text            = "Demo 2: Add Language for Application Already Installed"
   next_type       = "n"

#----
# Since demo_mle_inst_lang_hdr is a descriptive, but not very
# memorable name, an alias with a simpler name can be made to
# point to the same place.
#----
sm_menu_opt:
   id              = "demo_lang"
   next_id         = "demo_mle_inst_lang_hdr"
   next_type       = "n"
   alias           = "y"

sm_menu_opt:
   id_seq_num      = "030"
   id              = "demo"
   next_id         = "demo_lspv"
   text            = "Demo 3: List Contents of a Physical Volume"
   text_msg_file   = "smit.cat"
   next_type       = "n"

sm_menu_opt:
   id_seq_num      = "040"
   id              = "demo"
   next_id         = "demo_date"
   text            = "Demo 4: Change / Show Date, Time"
   text_msg_file   = "smit.cat"
   next_type       = "n"

#------------------------------------------------------------------
# Demo 1
# ------
# Goal: Add a Print Queue. If the printers.rte package is not
#       installed, install it automatically. If the user is
#       running MSMIT (SMIT in a windows interface), launch a
#       graphical program for this task. Otherwise, branch to
#       the SMIT print queue task.
#
# Topics:      1. cooked output & cmd_to_classify
#              2. SMIT environment variable (msmit vs. ascii)
#              3. ghost name_hdr
#              4. super-ghost name_hdr
#              5. creating an "OK / cancel" option
#              6. dspmsg for translations
#              7. exit/exec mode
#              8. id_seq_num for a name_hdr option
#------------------------------------------------------------------
#----
# Topics: 1,4
# Note that the next_id is the same as the id. Remember that the
# output of the cmd_to_classify is appended to the next_id,
# since the type is "c", for cooked. So, the next_id will be
# either demo_queue1 or demo_queue2. None of the output of the
# name_hdr is displayed, and there is no cmd_to_list in the
# demo_queue_dummy_opt, making this name_hdr a super-ghost.
#----
sm_name_hdr:
   id                      = "demo_queue"
   next_id                 = "demo_queue"
   option_id               = "demo_queue_dummy_opt"
   name                    = "Add a Print Queue"
   name_msg_file           = "smit.cat"
   name_msg_set            = 52
   name_msg_id             = 41
   type                    = "c"
   ghost                   = "y"
```

```
   cmd_to_classify         = "\
x()
{
    # Check to see if the printer file is installed.
    lslpp -l printers.rte 2>/dev/null 1>/dev/null
    if [[ $? != 0 ]]
    then
   echo 2
    else
   echo 1
    fi
}
x"
   next_type                  = "n"

#----
# Topics: 2,4
# Having determined the printer software is installed, we want
# to know if the gui program should be run or if we should
# branch to the ascii SMIT screen for this task. To do this, we
# check the value of the environment variable SMIT, which is "m"
# for windows (Motif) or "a" for ascii. Here again we tack the
# output of the cmd_to_classify onto the next_id.
#----
sm_name_hdr:
   id                         = "demo_queue1"
   next_id                    = "mkpq"
   option_id                  = "demo_queue_dummy_opt"
   has_name_select            = ""
   ghost                      = "y"
   next_type                  = "n"
   type                       = "c"
   cmd_to_classify            = "\
gui_check()
{
   if [ $SMIT = \"m\" ]; then
     echo gui
   fi
}
   gui_check"

sm_name_hdr:
   id               = "mkpqgui"
   next_id          = "invoke_gui"
   next_type        = "d"
   option_id        = "demo_queue_dummy_opt"
   ghost            = "y"

#----
# Topics: 7
# Note: the exec_mode of this command is "e", which
# exits SMIT before running the cmd_to_exec.
#----
sm_cmd_hdr:
   id               = "invoke_gui"
   cmd_to_exec      = "/usr/bin/X11/xprintm"
   exec_mode        = "e"
   ghost            = "y"

sm_cmd_opt:
   id                             = "demo_queue_dummy_opt"
   id_seq_num                     = 0

#----
# Topics: 3,5
# The printer software is not installed. Install the software
# and loop back to demo_queue1 to check the SMIT environment
```

```
# variable. This is a ghost name_hdr. The cmd_to_list of the
# sm_cmd_opt is displayed immediately as a pop-up option
# instead of waiting for the user to input a response. In this
# ghost, the cmd_opt is a simple OK/cancel box that prompts the
# user to press return.
#----
sm_name_hdr:
   id                        = "demo_queue2"
   next_id                   = "demo_queue1"
   option_id                 = "demo_queue_opt"
   name                      = "Add a Print Queue"
   name_msg_file             = "smit.cat"
   name_msg_set              = 52
   name_msg_id               = 41
   ghost                     = "y"
   cmd_to_classify           = "\
install_printers ()
{

  # Install the printer package.
  /usr/lib/assist/install_pkg \"printers.rte\" 2>&1 >/dev/null
  if [[ $? != 0 ]]
  then
    echo "Error installing printers.rte"
    exit 1
  else
    exit 0
  fi
}
install_printers "
   next_type                        = "n"

#----
# Topics: 5,6,8
# Here a cmd_opt is used as an OK/cancel box. Note also that the
# command dspmsg is used to display the text for the option. This
# allows for translation of the messages.
# Note: the id_seq_num for the option is 0. Only one option is
#       allowed per name_hdr, and its id_seq_num must be 0.
#----
sm_cmd_opt:
   id                        = "demo_queue_opt"
   id_seq_num                = "0"
   disc_field_name           = ""
   name                      = "Add a Print Queue"
   name_msg_file             = "smit.cat"
   name_msg_set              = 52
   name_msg_id               = 41
   op_type                   = "l"
   cmd_to_list               = "x()\
{
if [ $SMIT = \"a\" ] \n\
then \n\
dspmsg -s 52 smit.cat 56 \
'Press Enter to automatically install the printer software.\n\
Press F3 to cancel.\n\
'\n\
else \n\
dspmsg -s 52 smit.cat 57 'Click on this item to automatically install
the printer software.\n' \n\
fi\n\
} \n\
x"
   entry_type                = "t"
   multi_select              = "n"

#------------------------------------------------------------------
```

```
#
# Demo 2
# ------
# Goal: Add a Language for an Application Already Installed. It
#       is often clearer to the user to get some information
#       before displaying the dialog screen. Name Headers
#       (sm_name_hdr) can be used for this purpose. In this
#       example, two name headers are used to determine the
#       language to install and the installation device. The
#       dialog has entries for the rest of the information needed
#       to perform the task.
#
# Topics:
#       1. Saving output from successive name_hdrs with
#          cooked_field_name
#       2. Using getopts inside cmd_to_exec to process cmd_opt
#          info
#       3. Ring list vs. cmd_to_list for displaying values
#          cmd_opts
#-------------------------------------------------------------------

#----
# Topic: 1
# This is the first name_hdr. It is called by the menu_opt for
# this function. We want to save the user's input for later use
# in the dialog. The parameter passed into the cmd_to_classify
# comes from the user's selection/entry. Cmd_to_classify cleans
# up the output and stores it in the variable specified by
# cooked_field_name. This overrides the default value for the
# cmd_to_classify output, which is _cookedname. The default must
# be overridden because we also need to save the output of the
# next name_hdr.
#----
sm_name_hdr:
    id                      = "demo_mle_inst_lang_hdr"
    next_id                 = "demo_mle_inst_lang"
    option_id               = "demo_mle_inst_lang_select"
    name                    = "Add Language for Application Already Installed"
    name_msg_file           = "smit.cat"
    name_msg_set            = 53
    name_msg_id             = 35
    type                    = "j"
    ghost                   = "n"
    cmd_to_classify         = "\
        foo() {
            echo $1 | sed -n \"s/[^[]*\\[\\([^]]*\\).*/\\1/p\"
        }
        foo"
    cooked_field_name       = "add_lang_language"
    next_type               = "n"
    help_msg_id             = "2850325"

sm_cmd_opt:
    id                      = "demo_mle_inst_lang_select"
    id_seq_num              = "0"
    disc_field_name         = "add_lang_language"
    name                    = "LANGUAGE translation to install"
    name_msg_file           = "smit.cat"
    name_msg_set            = 53
    name_msg_id             = 20
    op_type                 = "l"
    entry_type              = "n"
    entry_size              = 0
    required                = ""
    prefix                  = "-l "
    cmd_to_list_mode        = "a"
    cmd_to_list             = "/usr/lib/nls/lsmle -l"
```

```
    help_msg_id               = "2850328"

#----
# Topic:1
# This is the second name_hdr. Here the user's input is passed
# directly through the cmd_to_classify and stored in the
# variable add_lang_input.
#----
sm_name_hdr:
    id                        = "demo_mle_inst_lang"
    next_id                   = "demo_dialog_add_lang"
    option_id                 = "demo_add_input_select"
    has_name_select           = "y"
    name                      = "Add Language for Application Already Installed"
    name_msg_file             = "smit.cat"
    name_msg_set              = 53
    name_msg_id               = 35
    type                      = "j"
    ghost                     = "n"
    cmd_to_classify           = "\
        foo() {
            echo $1
        }
        foo"
    cooked_field_name         = "add_lang_input"
    next_type                 = "d"
    help_msg_id               = "2850328"

sm_cmd_opt:
    id                        = "demo_add_input_select"
    id_seq_num                = "0"
    disc_field_name           = "add_lang_input"
    name                      = "INPUT device/directory for software"
    name_msg_file             = "smit.cat"
    name_msg_set              = 53
    name_msg_id               = 11
    op_type                   = "l"
    entry_type                = "t"
    entry_size                = 0
    required                  = "y"
    prefix                    = "-d "
    cmd_to_list_mode          = "1"
    cmd_to_list               = "/usr/lib/instl/sm_inst list_devices"
    help_msg_id               = "2850313"

#----
# Topic: 2
# Each of the cmd_opts formats its information for processing
# by the getopts command (a dash and a single character, followed
# by an optional parameter). The colon following the letter in
# the getopts command means that a parameter is expected after
# the dash option. This is a nice way to process the cmd_opt
# information if there are several options, especially if one of
# the options could be left out, causing the sequence of $1, $2,
# etc. to get out of order.
#----
sm_cmd_hdr:
    id              = "demo_dialog_add_lang"
    option_id       = "demo_mle_add_app_lang"
    has_name_select = ""
    name      = "Add Language for Application  Already Installed"
    name_msg_file   = "smit.cat"
    name_msg_set    = 53
    name_msg_id     = 35
    cmd_to_exec     = "\
       foo()
       {
```

```
       while getopts d:l:S:X Option \"$@\"
       do
          case $Option in
             d) device=$OPTARG;;
             l) language=$OPTARG;;
             S) software=$OPTARG;;
             X) extend_fs="-X";;
          esac
       done

       if [[ '/usr/lib/assist/check_cd -d $device' = '1' ]]
       then
          /usr/lib/assist/mount_cd $device
          CD_MOUNTED=true
       fi

       if [[ $software = \"ALL\" ]]
       then
          echo "Installing all software for $language..."
       else
          echo "Installing $software for $language..."
       fi
       exit $RC
       }
       foo"
    ask                     = "y"
    ghost                   = "n"
    help_msg_id             = "2850325"

sm_cmd_opt:
    id                      = "demo_mle_add_app_lang"
    id_seq_num              = "0"
    disc_field_name         = "add_lang_language"
    name                    = "LANGUAGE translation to install"
    name_msg_file           = "smit.cat"
    name_msg_set            = 53
    name_msg_id             = 20
    entry_type              = "n"
    entry_size              = 0
    required                = "y"
    prefix                  = "-l "
    cmd_to_list_mode        = "a"
    help_msg_id             = "2850328"

#----
# Topic: 2
# The prefix field precedes the value selected by the user, and
# both the prefix and the user-selected value are passed into
# the cmd_to_exec for getopts processing.
#----
sm_cmd_opt:
    id                  = "demo_mle_add_app_lang"
    id_seq_num          = "020"
    disc_field_name     = "add_lang_input"
    name                = "INPUT device/directory for software"
    name_msg_file       = "smit.cat"
    name_msg_set        = 53
    name_msg_id         = 11
    entry_type          = "n"
    entry_size          = 0
    required            = "y"
    prefix              = "-d "
    cmd_to_list_mode    = "1"
    cmd_to_list         = "/usr/lib/instl/sm_inst list_devices"
    help_msg_id         = "2850313"

sm_cmd_opt:
```

```
    id                    = "demo_mle_add_app_lang"
    id_seq_num            = "030"
    name                  = "Installed APPLICATION"
    name_msg_file         = "smit.cat"
    name_msg_set          = 53
    name_msg_id           = 43
    op_type               = "l"
    entry_type            = "n"
    entry_size            = 0
    required              = "y"
    prefix                = "-S "
    cmd_to_list_mode      = ""
    cmd_to_list           = "\
       list_messages ()
       {
          language=$1
          device=$2
          lslpp -Lqc | cut -f2,3 -d':'
       }
       list_messages"
    cmd_to_list_postfix   = "add_lang_language add_lang_input"
    multi_select          = ","
    value_index           = 0
    disp_values           = "ALL"
    help_msg_id           = "2850329"

#----
# Topic: 3
# Here, instead of a cmd_to_list, there is a comma-delimited set
# of Ring values in the disp_values field. This list is displayed
# one item at a time as the user presses tab in the cmd_opt entry
# field. However, instead of passing a yes or no to the cmd_hdr,
# it is more useful to use the aix_values field to pass either
# a -X or nothing. The list in the aix_values field must match
# one-to-one with the list in the disp_values field.
#----
sm_cmd_opt:
   id_seq_num = "40"
   id = "demo_mle_add_app_lang"
   disc_field_name = ""
   name = "EXTEND file systems if space needed?"
   name_msg_file = "smit.cat"
   name_msg_set = 53
   name_msg_id = 12
   op_type = "r"
   entry_type = "n"
   entry_size = 0
   required = "y"
   multi_select = "n"
   value_index = 0
   disp_values = "yes,no"
        values_msg_file = "sm_inst.cat"
   values_msg_set = 1
   values_msg_id = 51
   aix_values = "-X,"
   help_msg_id = "0503005"

#-------------------------------------------------------------------
#
# Demo 3
# ------
# Goal: Show Characteristics of a Logical Volume. The name of the
#   logical volume is entered by the user and passed to the
#   cmd_hdr as _rawname.
#
# Topics:       1. _rawname
#               2. Ringlist & aix_values
```

```
#------------------------------------------------------------

#----
# Topic: 1
# No rawname is needed because we have only one name_hdr and
# we can use the default variable name _rawname.
#----
sm_name_hdr:
   id = "demo_lspv"
   next_id = "demo_lspvd"
   option_id = "demo_cmdlvmpvns"
   has_name_select = ""
   name = "List Contents of a Physical Volume"
   name_msg_file = "smit.cat"
   name_msg_set = 15
   name_msg_id = 100
   type = "j"
   ghost = ""
   cmd_to_classify = ""
   raw_field_name = ""
   cooked_field_name = ""
   next_type = "d"
   help_msg_id = "0516100"

sm_cmd_opt:
   id_seq_num = "0"
   id = "demo_cmdlvmpvns"
   disc_field_name = "PVName"
   name = "PHYSICAL VOLUME name"
   name_msg_file = "smit.cat"
   name_msg_set = 15
   name_msg_id = 101
   op_type = "l"
   entry_type = "t"
   entry_size = 0
   required = "+"
   cmd_to_list_mode = "1"
   cmd_to_list = "lsvg -o|lsvg -i -p|grep -v '[:P]'| \
      cut -f1 -d' '"
   cmd_to_list_postfix = ""
   multi_select = "n"
   help_msg_id = "0516021"

#----
# Topic: 1
# The cmd_to_discover_postfix passes in the name of the physical
# volume, which is the raw data selected by the user in the
# name_hdr - _rawname.
#----
sm_cmd_hdr:
   id = "demo_lspvd"
   option_id = "demo_cmdlvmlspv"
   has_name_select = "y"
   name = "List Contents of a Physical Volume"
   name_msg_file = "smit.cat"
   name_msg_set = 15
   name_msg_id = 100
   cmd_to_exec = "lspv"
   ask = "n"
   cmd_to_discover_postfix = "_rawname"
   help_msg_id = "0516100"

sm_cmd_opt:
   id_seq_num = "01"
   id = "demo_cmdlvmlspv"
   disc_field_name = "_rawname"
   name = "PHYSICAL VOLUME name"
```

```
   name_msg_file = "smit.cat"
   name_msg_set = 15
   name_msg_id = 101
   op_type = "l"
   entry_type = "t"
   entry_size = 0
   required = "+"
   cmd_to_list_mode = "1"
   cmd_to_list = "lsvg -o|lsvg -i -p|grep -v '[:P]'| \
      cut -f1 -d' '"
   help_msg_id = "0516021"

#----
# Topic: 2
# Here a ringlist of 3 values matches with the aix_values we
# want to pass to the sm_cmd_hdr's cmd_to_exec.
#----
sm_cmd_opt:
   id_seq_num = "02"
   id = "demo_cmdlvmlspv"
   disc_field_name = "Option"
   name = "List OPTION"
   name_msg_file = "smit.cat"
   name_msg_set = 15
   name_msg_id = 92
   op_type = "r"
   entry_type = "n"
   entry_size = 0
   required = "n"
   value_index = 0
   disp_values = "status,logical volumes,physical \
           partitions"
   values_msg_file = "smit.cat"
   values_msg_set = 15
   values_msg_id = 103
   aix_values = "  ,-l,-p"
   help_msg_id = "0516102"

#-------------------------------------------------------------------
#
# Demo 4
# ------
# Goal: Change / Show Date & Time
#
# Topics:       1. Using a ghost name header to get variable
#                  values for the next dialog screen.
#               2. Using a cmd_to_discover to fill more than one
#                  cmd_opt with initial values.
#               3. Re-ordering parameters in a cmd_to_exec.
#-------------------------------------------------------------------

#----
# Topic: 1
# This ghost name_hdr gets two values and stores them in the
# variables daylight_y_n and time_zone for use in the cmd_opts
# for the next dialog. The output of cmd_to_classify is colon-
# delimited, as is the list of field names in cooked_field_name.
#----
sm_name_hdr:
   id = "demo_date"
   next_id = "demo_date_dial"
   option_id = "date_sel_opt"
   name_msg_set = 0
   name_msg_id = 0
   ghost = "y"
   cmd_to_classify = " \
if [ $(echo $TZ | awk '{ \
```

```
  if (length($1) <=6 ) {printf(\"2\")} \
  else {printf(\"1\")} }') = 1 ] \n\
then\n\
    echo $(dspmsg smit.cat -s 30 18 'yes')\":$TZ\"\n\
else\n\
    echo $(dspmsg smit.cat -s 30 19 'no')\":$TZ\"\n\
fi #"
   cooked_field_name = "daylight_y_n:time_zone"

sm_cmd_opt:
   id_seq_num = "0"
   id = "date_sel_opt"

#----
# Topic: 2,3
# Here the cmd_to_discover gets six values, one for each of the
# editable sm_cmd_opts for this screen. The cmd_to_discover
# output is two lines, the first with a # followed by a list of
# variable names, and the second line the list of values. Both
# lists are colon-delimited. We also see here the cmd_to_exec
# takeing the parameters from the cmd_opts and reordering them
# when calling the command.
#----
sm_cmd_hdr:
   id = "demo_date_dial"
   option_id = "demo_chtz_opts"
   has_name_select = "y"
   name = "Change / Show Date & Time"
   name_msg_file = "smit.cat"
   name_msg_set = 30
   name_msg_id = 21
   cmd_to_exec = "date_proc () \
# MM dd hh mm ss yy\n\
# dialogue param order   #  3  4  5  6  7  2\n\
{\n\
date \"$3$4$5$6.$7$2\"\n\
}\n\
date_proc "
   exec_mode = "P"
   cmd_to_discover = "disc_proc() \n\
{\n\
TZ=\"$1\"\n\
echo '#cur_month:cur_day:cur_hour:cur_min:cur_sec:cur_year'\n\
date +%m:%d:%H:%M:%S:%y\n\
}\n\
disc_proc"
   cmd_to_discover_postfix = ""
   help_msg_id = "055101"

#----
# The first two cmd_opts get their initial values
# (disc_field_name) from the name_hdr.
#----
sm_cmd_opt:
   id_seq_num = "04"
   id = "demo_chtz_opts"
   disc_field_name = "time_zone"
   name = "Time zone"
   name_msg_file = "smit.cat"
   name_msg_set = 30
   name_msg_id = 16
   required = "y"

sm_cmd_opt:
   id_seq_num = "08"
   id = "demo_chtz_opts"
   disc_field_name = "daylight_y_n"
```

```
   name = "Does this time zone go on daylight savings time?\n"
   name_msg_file = "smit.cat"
   name_msg_set = 30
   name_msg_id = 17
   entry_size = 0

#----
# The last six cmd_opts get their values from the
# cmd_to_discover.
#----
sm_cmd_opt:
   id_seq_num = "10"
   id = "demo_chtz_opts"
   disc_field_name = "cur_year"
   name = "YEAR (00-99)"
   name_msg_file = "smit.cat"
   name_msg_set = 30
   name_msg_id = 10
   entry_type = "#"
   entry_size = 2
   required = "+"
   help_msg_id = "055102"

sm_cmd_opt:
   id_seq_num = "20"
   id = "demo_chtz_opts"
   disc_field_name = "cur_month"
   name = "MONTH (01-12)"
   name_msg_file = "smit.cat"
   name_msg_set = 30
   name_msg_id = 11
   entry_type = "#"
   entry_size = 2
   required = "+"
   help_msg_id = "055132"

sm_cmd_opt:
   id_seq_num = "30"
   id = "demo_chtz_opts"
   disc_field_name = "cur_day"
   name = "DAY (01-31)\n"
   name_msg_file = "smit.cat"
   name_msg_set = 30
   name_msg_id = 12
   entry_type = "#"
   entry_size = 2
   required = "+"
   help_msg_id = "055133"

sm_cmd_opt:
   id_seq_num = "40"
   id = "demo_chtz_opts"
   disc_field_name = "cur_hour"
   name = "HOUR (00-23)"
   name_msg_file = "smit.cat"
   name_msg_set = 30
   name_msg_id = 13
   entry_type = "#"
   entry_size = 2
   required = "+"
   help_msg_id = "055134"

sm_cmd_opt:
   id_seq_num = "50"
   id = "demo_chtz_opts"
   disc_field_name = "cur_min"
   name = "MINUTES (00-59)"
```

```
    name_msg_file = "smit.cat"
    name_msg_set = 30
    name_msg_id = 14
    entry_type = "#"
    entry_size = 2
    required = "+"
    help_msg_id = "055135"

sm_cmd_opt:
    id_seq_num = "60"
    id = "demo_chtz_opts"
    disc_field_name = "cur_sec"
    name = "SECONDS (00-59)"
    name_msg_file = "smit.cat"
    name_msg_set = 30
    name_msg_id = 15
    entry_type = "#"
    entry_size = 2
    required = "+"
    help_msg_id = "055136"
```

Chapter 26. System Resource Controller

This article provides information about the System Resource Controller (SRC), which facilitates the management and control of complex subsystems.

The SRC is a subsystem controller. Subsystem programmers who own one or more daemon processes can use SRC services to define a consistent system management interface for their applications. The SRC provides a single set of commands to start, stop, trace, refresh, and query the status of a subsystem.

In addition, the SRC provides an error notification facility. You can use this facility to incorporate subsystem-specific recovery methods. The type of recovery information included is limited only by the requirement that the notify method is a string in a file and is executable.

Refer to the following information to learn more about SRC programming requirements:

Subsystem Interaction with the SRC

The SRC defines a subsystem as a program or set of related programs designed as a unit to perform related functions. See ″System Resource Controller Overview″ in *AIX 5L Version 5.1 System Management Guide: Operating System and Devices* for a more detailed description of the characteristics of a subsystem.

A subserver, commonly known to UNIX programmers as a daemon, is a process that belongs to and is controlled by a subsystem.

The SRC operates on objects in the SRC object class. Subsystems are defined to the SRC as subsystem objects; subservers, as subserver-type objects. The structures associated with each type of object are predefined in the **usr/include/sys/srcobj.h** file.

The SRC can issue SRC commands against objects at the subsystem, subserver, and subsystem-group levels. A subsystem group is a group of any user-specified subsystems. Grouping subsystems allows multiple subsystems to be controlled by invoking a single command. Groups of subsystems may also share a common notification method.

The SRC communicates with subsystems by sending signals and exchanging request and reply packets. In addition to signals, the SRC recognizes the sockets and IPC message-queue communication types. A number of subroutines ("List of Additional SRC Subroutines" on page 712) are available as an SRC API to assist in programming communication between subsystems and the SRC. The SRC API also supports programming communication between client programs and the SRC.

The SRC and the init Command

The SRC is operationally independent of the **init** command. However, the SRC is intended to extend the process-spawning functionality provided by this command. In addition to providing a single point of control to start, stop, trace, refresh, and query the status of subsystems, the SRC can control the operations of individual subsystems, support remote system control, and log subsystem failures.

Operationally, the only time the **init** command and the SRC interact occurs when the **srcmstr** (SRC master) daemon is embedded within the **inittab** file. (By default, the **srcmstr** daemon is in the **inittab** file.)

In this case, the **init** command starts the **srcmstr** daemon at system startup, as with all other processes. You must have root user authority or be in the system group to invoke the **srcmstr** daemon.

Compiling Programs to Interact With the srcmstr Daemon

To enable programs to interact with the **srcmstr** daemon, the **/usr/include/spc.h** file should be included and the program should be compiled with the **libsrc.a** library. This support is not needed if the subsystem uses signals to communicate with the SRC.

SRC Operations

To make use of SRC functionality, a subsystem must interact with the **srcmstr** daemon in two ways:

- A subsystem object must be created for the subsystem in the SRC subsystem object class.
- If a subsystem uses signals, it does not need to use SRC subroutines. However, if it uses message queues or sockets, it must respond to stop requests using the SRC subroutines.

All SRC subsystems must support the **stopsrc** command. The SRC uses this command to stop subsystems and their subservers with the **SIGNORM** (stop normal), **SIGFORCE** (stop force), or **SIGCANCEL** (cancel systems) signals.

Subsystem support is optional for the **startsrc**, **lssrc -l**, **traceson**, **tracesoff**, and **refresh** commands, long status and subserver status reporting, and the SRC notification mechanism. See “Programming Subsystem Communication with the SRC” on page 705 for details.

SRC Capabilities

The SRC provides the following support for the subsystem programmer:

- A common command interface to support starting, stopping, and sending requests to a subsystem
- A central point of control for subsystems and groups of subsystems
- A common format for requests to the subsystem
- A definition of subservers so that each subserver can be managed as it is uniquely defined to the subsystem
- The ability to define subsystem-specific error notification methods
- The ability to define subsystem-specific responses to requests for status, trace support, and configuration refresh
- A single point of control for servicing subsystem requests in a network computing environment

SRC Objects

The System Resource Controller (SRC) defines and manages three object classes:

- “Subsystem Object Class” on page 699
- “Subserver Type Object Class” on page 701
- “Notify Object Class” on page 701

Together, these object classes represent the domain in which the SRC performs its functions. A predefined set of object-class descriptors comprise the possible set of subsystem configurations supported by the SRC.

> **Note:** Only the SRC Subsystem object class is required. Use of the Subserver Type and Notify object classes is subsystem-dependent.

Subsystem Object Class

The subsystem object class contains the descriptors for all SRC subsystems. A subsystem must be configured in this class before it can be recognized by the SRC.

The descriptors for the Subsystem object class are defined in the **SRCsubsys** structure of the **/usr/include/sys/srcobj.h** file. The Subsystem Object Descriptors and Default Values table provides a short-form illustration of the subsystem descriptors as well as the **mkssys** and **chssys** command flags associated with each descriptor.

Subsystem Object Descriptors and Default Values		
Descriptors	**Default Values**	**Flags**
Subsystem name		-s
Path to subsystem command		-p
Command arguments		-a
Execution priority	20	-E
Multiple instance	NO	-Q -q
User ID		-u
Synonym name (key)		-t
Start action	ONCE	-O -R
stdin	/dev/console	-i
stdout	/dev/console	-o
stderr	/dev/console	-e
Communication type	Sockets	-K -I -S
Subsystem message type		-m
Communication IPC queue key		-l
Group name		-G
SIGNORM signal		-n
SIGFORCE signal		-f
Display	Yes	-D -d
Wait time	20 seconds	-w
Auditid		

The subsystem object descriptors are defined as follows:

Subsystem name Specifies the name of the subsystem object. The name cannot exceed 30 bytes, including the null terminator (29 characters for single-byte character sets, or 14 characters for multibyte character sets). This descriptor must be POSIX-compliant. This field is required.

Subsystem command path Specifies the full path name for the program executed by the subsystem start command. The path name cannot exceed 200 bytes, including the null terminator (199 characters for single-byte character sets, or 99 characters for multibyte character sets). The path name must be POSIX-compliant. This field is required.

Command arguments Specifies any arguments that must be passed to the command that starts the subsystem. The arguments cannot exceed 200 bytes, including the null terminator (199 characters for single-byte character sets, or 99 characters for multibyte character sets). The arguments are parsed by the **srcmstr** daemon according to the same rules used by shells. For example, quoted strings are passed as a single argument, and blanks outside quoted strings delimit arguments.

Execution priority Specifies the process priority of the subsystem to be run. Subsystems started by the **srcmstr** daemon run with this priority. The default value is 20.

Multiple instance Specifies the number of instances of a subsystem that can run at one time. A value of NO (the **-Q** flag) specifies that only one instance of the subsystem can run at one time. Attempts to start this subsystem if it is already running will fail, as will attempts to start a subsystem on the same IPC message queue key. A value of YES (the **-q** flag) specifies that multiple subsystems may use the same IPC message queue and that there can be multiple instances of the same subsystem. The default value is NO.

User ID Specifies the user ID (numeric) under which the subsystem is run. A value of 0 indicates the root user.

Synonym name Specifies a character string to be used as an alternate name for the subsystem. The character string cannot exceed 30 bytes, including the null terminator (29 characters for single-byte character sets, or 14 characters for multibyte character sets). This field is optional.

Start action Specifies whether the **srcmstr** daemon should restart the subsystem after an abnormal end. A value of RESPAWN (the **-R** flag) specifies the **srcmstr** daemon should restart the subsystem. A value of ONCE (the **-O** flag) specifies the **srcmstr** daemon should not attempt to restart the failed system. There is a respawn limit of two restarts within a specified wait time. If the failed subsystem cannot be successfully restarted, the notification method option is consulted. The default value is ONCE.

Standard Input File/Device Specifies the file or device from which the subsystem receives its input. The default is **/dev/console**. This field cannot exceed 200 bytes, including the null terminator (199 characters for single-byte character sets, or 99 characters for multibyte character sets). This field is ignored if the communication type is sockets.

Standard Output File/Device Specifies the file or device to which the subsystem sends its output. This field cannot exceed 200 bytes, including the null terminator (199 characters for single-byte character sets, or 99 characters for multibyte character sets). The default is **/dev/console**.

Standard Error File/Device Specifies the file or device to which the subsystem writes its error messages. This field cannot exceed 200 bytes, including the null terminator (199 characters for single-byte character sets, or 99 characters for multibyte character sets). Failures are handled as part of the notify method. The default is **/dev/console**.

Note: Catastrophic errors are sent to the error log.

Communication type Specifies the communication method between the **srcmstr** daemon and the subsystem. Three types can be defined: IPC (**-I**), sockets (**-K**), or signals (**-S**). The default is sockets.

Communication IPC queue key Specifies a decimal value that corresponds to the IPC message queue key that the **srcmstr** daemon uses to communicate to the subsystem. This field is required for subsystems that communicate using IPC message queues. Use the **ftok** subroutine with a fully qualified path name and an ID parameter to ensure that this key is unique. The **srcmstr** daemon creates the message queue prior to starting the subsystem.

Group name Designates the subsystem as a member of a group. This field cannot exceed 30 bytes, including the null terminator (29 characters for single-byte character sets, or 14 characters for multibyte character sets). This field is optional.

Subsystem message type	Specifies the mtype of the message that is placed on the subsystem's message queue. The subsystem uses this value to retrieve messages by using the **msgrcv** or **msgxrcv** subroutine. This field is required if you are using message queues.
SIGNORM signal value	Specifies the value to be sent to the subsystem when a stop normal request is sent. This field is required of subsystems using the signals communication type.
SIGFORCE signal value	Specifies the value to be sent to the subsystem when a stop force request is sent. This field is required of subsystems using the signals communication type.
Display value	Indicates whether the status of an inoperative subsystem can be displayed on **lssrc -a** or **lssrc -g** output. The **-d** flag indicates display; the **-D** flag indicates do not display. The default is **-d** (display).
Wait time	Specifies the time in seconds that a subsystem has to complete a restart or stop request before alternate action is taken. The default is 20 seconds.
Auditid	Specifies the subsystem audit ID. Created automatically by the **srcmstr** daemon when a subsystem is defined, this field is used by the security system, if configured. This field cannot be set or changed by a program.

See "Defining Your Subsystem to the SRC" on page 711 for information on defining and modifying subsystem objects.

Subserver Type Object Class

An object must be configured in this class if a subsystem has subservers and the subsystem expects to receive subserver-related commands from the **srcmstr** daemon.

This object class contains three descriptors, which are defined in the **SRCsubsvr** structure of the **srcobj.h** file:

Subserver ID (key)	Specifies the name of the subserver type object identifier. The set of subserver type names defines the allowable values for the **-t** flag of the subserver commands. The name length cannot exceed 30 bytes, including the terminating null (29 characters for single-byte character sets, or 14 characters for multibyte character sets).
Owning subsystem name	Specifies the name of the subsystem that owns the subserver object. This field is defined as a link to the SRC subsystem object class.
Code point	Specifies a decimal number that identifies the subserver. The code point is passed to the subsystem controlling the subserver in the `object` field of the **subreq** structure of the SRC request structure. If a subserver object name is also provided in the command, the **srcmstr** daemon forwards the code point to the subsystem in the `objname` field of the **subreq** structure. See the ″SRC Request Structure Example″ in the **spc.h** file documentation for examples of these elements.

The commands that reference subservers identify each subserver as a named type of subserver and can also append a name to each instance of a subserver type. The SRC daemon uses the subserver type to determine the controlling subsystem for the subserver, but does not examine the subserver name.

See "Defining Your Subsystem to the SRC" on page 711 for information on defining and modifying subserver type objects.

Notify Object Class

This class provides a mechanism for the **srcmstr** daemon to invoke subsystem-provided routines when the failure of a subsystem is detected. When the SRC daemon receives a **SIGCHLD** signal indicating the termination of a subsystem process, it checks the status of the subsystem (maintained by the **srcmstr**

daemon) to determine if the termination was caused by a **stopsrc** command. If no **stopsrc** command was issued, the termination is interpreted as an abnormal termination. If the restart action in the definition does not specify respawn, or if respawn attempts fail, the **srcmstr** daemon attempts to read an object associated with the subsystem name from the Notify object class. If such an object is found, the method associated with the subsystem is run.

If no subsystem object is found in the Notify object class, the **srcmstr** daemon determines whether the subsystem belongs to a group. If so, the **srcmstr** daemon attempts to read an object of that group name from the Notify object class. If such an object is found, the method associated with it is invoked. In this way, groups of subsystems can share a common method.

> **Note:** The subsystem notify method takes precedence over the group notify method. Therefore, a subsystem can belong to a group that is started together, but still have a specific recovery or cleanup routine defined.

Notify objects are defined by two descriptors:

Subsystem name or **Group name**	Specifies the name of the subsystem or group for which a notify method is defined.
Notify method	Specifies the full path name to the routine that is executed when the **srcmstr** daemon detects abnormal termination of the subsystem or group.

Such notification is useful when specific recovery or clean-up work needs to be performed before a subsystem can be restarted. It is also a tool for information gathering to determine why a subsystem abnormally stopped.

Notify objects are created with the **mknotify** command. To modify a notify method, the existing notify object must be removed using the **rmnotify** command, and then a new notify object created.

mknotify	Adds a notify method to the SRC configuration database
rmnotify	Removes a notify method from the SRC configuration database

The **srcmstr** daemon logs subsystem recovery activity. The subsystem is responsible for reporting subsystem failures.

SRC Communication Types

The System Resource Controller (SRC) supports three communication types: signals, sockets, and interprocess communication (IPC) message queues. The communication type chosen determines to what degree the subsystem takes advantage of SRC functions.

> **Note:** All subsystems, regardless of the communication type specified in the subsystem environment object, must be capable of supporting limited signals communication. A signal-catcher routine must be defined to handle **SIGTERM** (stop cancel) signals. The **SIGTERM** signal indicates a subsystem should clean up all resources and terminate.

The Communications Between the srcmstr Daemon and Subsystems table summarizes communication type actions associated with SRC functions.

Communications Between the srcmstr Daemon and Subsystems		
Function	**Using IPC or Sockets**	**Using Signals**
start		

subsystem	SRC forks and execs to create subsystem process.	SRC forks and execs to create subsystem process.
subserver	Uses IPC message queue or socket to send request to subsystem.	Not supported
stop normal		
subsystem	Uses IPC message queue or socket to send request to subsystem.	Sends **SIGNORM** to the subsystem.
subserver	Uses IPC message queue or socket to send request to subsystem.	Not supported.
stop forced		
subsystem	Uses IPC message queue or socket to send request to subsystem.	Sends **SIGFORCE** to the subsystem.
subserver	Uses IPC message queue or socket to send request to subsystem.	Not supported.
stop cancel		
subsystem	Sends **SIGTERM** followed by **SIGKILL** to the process group of the subsystem.	Sends **SIGTERM** followed by **SIGKILL** to the process group of the subsystem.
status short		
subsystem	Implemented by SRC (no subsystem request).	Implemented by SRC (no subsystem request).
subserver	Uses IPC message queue or socket to send request to subsystem.	Not supported.
status long		
subsystem	Uses IPC message queue or socket to send request to subsystem.	Not supported.
subserver	Uses IPC message queue or socket to send request to subsystem.	Not supported.
traceon/traceoff		
subsystem	Uses IPC message queue or socket to send request to subsystem.	Not supported.
subserver	Uses IPC message queue or socket to send request to subsystem.	Not supported.
refresh		
subsystem	Uses IPC message queue or socket to send request to subsystem.	Not supported.
subserver	Uses IPC message queue or socket to send request to subsystem.	Not supported.
notify		
subsystem	Implemented by subsystem-provided method.	Implemented by subsystem-provided method.

Signals Communication

The most basic type of communication between a subsystem and the **srcmstr** daemon is accomplished with signals. Because signals constitute a one-way communication scheme, the only SRC command that signals subsystems recognize is a stop request. Subsystems using signals do not recognize long status, refresh, or trace requests. Nor do they recognize subservers.

Signals subsystems must implement a signal-catcher routine, such as the **sigaction**, **sigvec**, or **signal** subroutine, to handle **SIGNORM** and **SIGFORCE** requests.

Signals subsystems are specified in the SRC subsystem object class by issuing a **mkssys -Snf** command string or by using the **defssys** and **addssys** subroutines.

addssys	Adds a subsystem definition to the SRC configuration database
defssys	Initializes a new subsystem definition with default values
mkssys	Adds a subsystem definition to the SRC configuration database

Sockets Communication

Increasingly, the communication option of choice for subsystem programmers is sockets. Sockets are also the default communication type for the **srcmstr** daemon. See the ″Sockets Overview″ in *AIX 5L Version 5.1 Communications Programming Concepts* for more information.

The **srcmstr** daemon uses sockets to receive work requests from a command process. When this communication type is chosen, the **srcmstr** daemon creates the subsystem socket in which the subsystem will receive **srcmstr** daemon requests. UNIX sockets (**AF_UNIX**) are created for local subsystems. Internet sockets (**AF_INET**) are created for remote subsystems. The following steps describe the command processing sequence:

1. The command process accepts a command from the input device, constructs a work-request message, and sends the work-request UDP datagram to the **srcmstr** daemon on the well-known SRC port. The **AF_INET** is identified in the **/etc/services** file.
2. The **srcmstr** daemon listens on the well-known SRC port for work requests. Upon receiving a work request, it tells the system to fill the **socket** subroutine's **sockaddr** structure to obtain the originating system's address and appends the address and port number to the work request.
3. The **srcmstr** daemon uses the **srcrrqs** and **srcsrpy** subroutines. It processes only those requests that it can process and then sends the information back to the command process. Other requests are forwarded to the appropriate subsystem on the port that the subsystem has specified for its work requests.
4. The subsystem listens on the port previously obtained by the **srcmstr** daemon for the subsystem. (Each subsystem inherits a port when the **srcmstr** daemon starts a subsystem.) The subsystem processes the work request and sends a reply back to the command process.
5. The command process listens for the response on the specified port.

The file access permissions and addresses of the sockets used by the **srcmstr** daemon are maintained in the **/dev/SRC** and **/dev/.SRC-unix** temporary directories. Though displayable using the **ls** command, the information contained in these directories is for internal SRC use only.

Message queues and sockets offer equal subsystem functionality.

See “Programming Subsystem Communication with the SRC” on page 705 for more information.

srcrrqs	Saves the destination address of your subsystem's response to a received packet. (Also see threadsafe version **srcrrqs_r**)
srcsrpy	Sends your subsystem response packet to a request that your subsystem received.

IPC Message Queue Communication

IPC message queue functionality is similar to sockets functionality. Both communication types support a full-function SRC environment.

When the communication type is IPC message queue, the **srcmstr** daemon uses sockets to receive work requests from a command process, then uses an IPC message queue in which the subsystem receives SRC messages. The message queue is created when the subsystem is started, and is used thereafter. Message queue subsystems use the following command-processing sequence to communicate with the **srcmstr** daemon:

1. The **srcmstr** daemon gets the message queue ID from the SRC subsystem object and sends the message to the subsystem.
2. The subsystem waits for the message queue and issues a **msgrcv** subroutine to receive the command from the message queue in the form of the **subreq** structure required of subsystem requests.
3. The subsystem calls the **srcrrqs** subroutine to get a tag ID that is used in responding to the message.
4. The subsystem interprets and processes the received command. Depending upon the command, the subsystem creates either a **svrreply** or **statcode** data structure to return a reply to the command process. Refer to the **/usr/include/spc.h** file for more information on these structures.
5. The subsystem calls the **srcsrpy** subroutine to send back a reply buffer to the command process.

See ″Message Queue Kernel Services″ in *AIX 5L Version 5.1 Kernel Extensions and Device Support Programming Concepts* for additional information on this communication type. See “Programming Subsystem Communication with the SRC” for the next step in establishing communication with the **srcmstr** daemon.

Programming Subsystem Communication with the SRC

System Resource Controller (SRC) commands are executable programs that take options from the command line. After the command syntax has been verified, the commands call SRC run-time subroutines to construct a User Datagram Protocol (UDP) datagram and send it to the **srcmstr** daemon.

The following sections provide more information about SRC subroutines and how they can be used by subsystems to communicate with the SRC main process:

Programming Subsystems to Receive SRC Requests

The programming tasks associated with receiving SRC requests vary with the communication type specified for the subsystem. The **srcmstr** daemon uses sockets to receive work requests from a command process and constructs the necessary socket or message queue to forward work requests. Each subsystem needs to verify the creation of its socket or message queue. See “SRC Communication Types” on page 702 for a description of the SRC communication types. Read the following sections for information on communication type-specific guidelines on programming your subsystem to receive SRC request packets.

Receiving SRC Signals

Subsystems that use signals as their communication type must define a signal-catcher routine to catch the **SIGNORM** and **SIGFORCE** signals. The signal-catching method used is subsystem-dependent. Following are two examples of the types of subroutines that can be used for this purpose.

sigaction, **sigvec**, or **signal** subroutine	Specifies the action to take upon the delivery of a signal.
sigset, **sighold**, **sigrelse**, or **sigignore** subroutine	Enhances the signal facility and provides signal management for application processes.

See “Signals Communication” on page 703 in ″Understanding SRC Communication Types″ for more information.

Receiving SRC Request Packets Using Sockets

Use the following guidelines when programming sockets subsystems to receive SRC request packets:

- Include the SRC subsystem structure in your subsystem code by specifying the **/usr/include/spc.h** file. This file contains the structures the subsystem uses to respond to SRC commands. In addition, the **spc.h** file includes the **srcerrno.h** file, which does not need to be included separately. The **srcerrno.h** file contains error-code definitions for daemon support.
- When a sockets subsystem is started, the socket on which the subsystem receives SRC request packets is set as file descriptor 0. The subsystem should verify this by calling the **getsockname** subroutine, which returns the address of the subsystem's socket. If file descriptor 0 is not a socket, the subsystem should log an error and then exit. See ″Reading Internet Datagrams Example Program″ in *AIX 5L Version 5.1 Communications Programming Concepts* for information on how the **getsockname** subroutine can be used to return the address of a subsystem socket.
- If a subsystem polls more than one socket, use the **select** subroutine to determine which socket has something to read. See ″Checking for Pending Connections Example Program″ in *AIX 5L Version 5.1 Communications Programming Concepts* for more information on how the **select** subroutine can be used for this purpose.
- Use the **recvfrom** subroutine to get the request packet from the socket.

  Note: The return address for the subsystem response packet is in the received SRC request packet. This address should not be confused with the address that the **recvfrom** subroutine returns as one of its parameters.

  After the **recvfrom** subroutine completes and the packet has been received, use the **srcrrqs** subroutine to return a pointer to a static **srchdr** structure. This pointer contains the return address for the subsystem's reply. This structure is overwritten each time the **srcrrqs** subroutine is called, so its contents should be stored elsewhere if they will be needed after the next call to the **srcrrqs** subroutine.

See "Programming Subsystems to Process SRC Request Packets" on page 707 for the next step in establishing subsystem communication with the SRC.

Receiving SRC Request Packets Using Message Queues

Use the following guidelines when programming message queue subsystems to receive SRC request packets:

- Include the SRC subsystem structure in your subsystem code by specifying the **/usr/include/spc.h** file. This file contains the structures the subsystem uses to respond to SRC commands. In addition, the **spc.h** file includes the **srcerrno.h** include file, which does not need to be included separately. The **srcerrno.h** file contains error-code definitions for daemon support.
- Specify **-DSRCBYQUEUE** as a compile option. This places a message type (`mtype`) field as the first field in the **srcreq** structure. This structure should be used any time an SRC packet is received.
- When the subsystem has been started, use the **msgget** subroutine to verify that a message queue was created at system startup. The subsystem should log an error and exit if a message queue was not created.
- If a subsystem polls more than one message queue, use the **select** subroutine to determine which message queue has something to read. See ″Checking for Pending Connections Example Program″ in *AIX 5L Version 5.1 Communications Programming Concepts* for information on how the **select** subroutine can be used for this purpose.
- Use the **msgrcv** or **msgxrcv** subroutine to get the packet from the message queue. The return address for the subsystem response packet is in the received packet.
- When the **msgrcv** or **msgxrcv** subroutine completes and the packet has been received, call the **srcrrqs** subroutine to finish the reception process. The **srcrrqs** subroutine returns a pointer to a static **srchdr** structure that is overwritten each time the **srcrrqs** subroutine is called. This pointer contains the return address for the subsystem's reply.

Programming Subsystems to Process SRC Request Packets

Subsystems must be capable of processing stop requests. Optionally, subsystems may support start, status, trace, and refresh requests.

Processing request packets involves a two-step process:

1. Reading SRC request packets
2. "Programming Subsystem Response to SRC Requests"

Reading SRC Request Packets

SRC request packets are received by subsystems in the form of a **srcreq** structure as defined in the **/usr/include/spc.h** file. The subsystem request resides in the **subreq** structure of the **srcreq** structure:

```
struct subreq
   short object;         /*object to act on*/
   short action;         /*action START, STOP, STATUS, TRACE,\
                           REFRESH*/
   short parm1;          /*reserved for variables*/
   short parm2;          /*reserved for variables*/
   char objname;         /*object name*/
```

The `object` field of the **subreq** structure indicates the object to which the request applies. When the request applies to a subsystem, the `object` field is set to the SUBSYSTEM constant. Otherwise, the `object` field is set to the subserver code point or the `objname` field is set to the subserver PID as a character string. It is the subsystem's responsibility to determine the object to which the request applies.

The `action` field specifies the action requested of the subsystem. Subsystems should understand the START, STOP, and STATUS action codes. The TRACE and REFRESH action codes are optional.

The `parm1` and `parm2` fields are used differently by each of the actions.

Action	parm1	parm2
STOP	NORMAL or FORCE	
STATUS	LONGSTAT or SHORTSTAT	
TRACE	LONGTRACE or SHORT-TRACE	TRACEON or TRACEOFF

The START subserver and REFRESH actions do not use the `parm1` and `parm2` fields.

Programming Subsystem Response to SRC Requests

The appropriate subsystem actions for the majority of SRC requests are programmed when the subsystem object is defined to the SRC. See "SRC Objects" on page 698 and "Defining Your Subsystem to the SRC" on page 711 for more information. The structures that subsystems use to respond to SRC requests are defined in the **/usr/include/spc.h** file. Subsystems may use the following SRC run-time subroutines to meet command processing requirements:

srcrrqs Allows a subsystem to store the header from a request.
srcsrpy Allows a subsystem to send a reply to a request.

See "Responding to Trace Requests" on page 710 and "Responding to Refresh Requests" on page 711 for information on how to program support for these commands in your subsystem.

Status-request processing requires a combination of tasks and subroutines. See "Processing SRC Status Requests" on page 708 for more information.

When subsystems receive requests they cannot process or that are invalid, they must send an error packet with an error code of **SRC_SUBICMD** in response to the unknown, or invalid, request. SRC reserves action codes 0-255 for SRC internal use. If your subsystem receives a request containing an action code that is not valid, your subsystem must return an error code of **SRC_SUBICMD**. Valid action codes supported by SRC are defined in the **spc.h** file. You can also define subsystem-specific action codes. An action code is not valid if it is not defined by the SRC or your subsystem. See "Programming Subsystems to Return SRC Error Packets" on page 710 for more information.

Note: Action codes 0-255 are reserved for SRC use.

Processing SRC Status Requests

Subsystems may be requested to provide three types of status reports: long subsystem status, short subserver status, and long subserver status.

Note: Short subsystem status reporting is performed by the **srcmstr** daemon. Statcode and reply-status value constants for this type of report are defined in the **/usr/include/spc.h** file. The Status Value Constants table lists required and suggested reply-status value codes.

Reply Status Value Codes			
Value	**Meaning**	**Subsystem**	**Subserver**
SRCWARN	Received a request to stop. (Will be stopped within 20 seconds.)	X	X
SRCACT	Started and active.	X	X
SRCINAC	Not active.		
SRCINOP	Inoperative.	X	X
SRCLOSD	Closed.		
SRCLSPN	In the process of being closed.		
SRCNOSTAT	Idle.		
SRCOBIN	Open, but not active.		
SRCOPND	Open.		
SRCOPPN	In the process of being opened.		
SRCSTAR	Starting.		X
SRCSTPG	Stopping.	X	X
SRCTST	TEST active.		
SRCTSTPN	TEST pending.		

The SRC **lssrc** command displays the received information on standard output. The information returned by subsystems in response to a long status request is left to the discretion of the subsystem. Subsystems that own subservers are responsible for tracking and reporting the state changes of subservers, if desired. Use the **srcstathdr** subroutine to retrieve a standard status header to pass back at the beginning of your status data.

The following steps are recommended in processing status requests:

1. To return status from a subsystem (short or long), allocate an array of **statcode** structures plus a **srchdr** structure. The **srchdr** structure must start the buffer that you are sending in response to the status request. The **statcode** structure is defined in the **/usr/include/spc.h** file.

```
struct statcode
{
   short objtype;
   short status;
   char objtext [65];
   char objname [30];
};
```

2. Fill in the `objtype` field with the SUBSYSTEM constant to indicate that the status is for a subsystem, or with a subserver code point to indicate that the status is for a subserver.
3. Fill in the `status` field with one of the SRC status constants defined in the **spc.h** file.
4. Fill in the `objtext` field with the NLS text that you wish displayed as status.
5. Fill in the `objname` field with the name of the subsystem or subserver for which the `objtext` field applies.

> **Note:** The subsystem and requester can agree to send other subsystem-defined information back to the requester. See "srcsrpy Continuation Packets" for more information on this type of response.

Programming Subsystems to Send Reply Packets

The packet that a subsystem returns to the SRC should be in the form of the **srcrep** structure as defined in the **/usr/include/spc.h** file. The **svrreply** structure that is part of the **srcrep** structure will contain the subsystem reply:

```
struct svrreply
{
   short rtncode;        /*return code from the subsystem*/
   short objtype;        /*SUBSYSTEM or SUBSERVER*/
   char objtext[65];     /*object description*/
   char objname[20];     /*object name*/
   char rtnmsg[256];     /*returned message*/
};
```

Use the **srcsrpy** subroutine to return a packet to the requester.

Creating a Reply

To program a subsystem reply, use the following procedure:

1. Fill in the `rtncode` field with the SRC error code that applies. Use **SRC_SUBMSG** as the `rtncode` field to return a subsystem-specific NLS message.
2. Fill in the `objtype` field with the SUBSYSTEM constant to indicate that the reply is for a subsystem, or with the subserver code point to indicate that the reply is for a subserver.
3. Fill in the `objname` field with the subsystem name, subserver type, or subserver object that applies to the reply.
4. Fill in the `rtnmsg` field with the subsystem-specific NLS message.
5. Key the appropriate entry in the **srcsrpy** *Continued* parameter. See ″srcsrpy Continuation Packets″ for more information.

> **Note:** The last packet from the subsystem must always have END specified in the *Continued* parameter to the **srcsrpy** subroutine.

srcsrpy Continuation Packets

Subsystem responses to SRC requests are made in the form of continuation packets. Two types of continuation packets may be specified: Informative message, and reply packets.

The informative message is not passed back to the client. Instead, it is printed to the client's standard output. The message must consist of NLS text, with message tokens filled in by the sending subsystem. To send this type of continuation packet, specify CONTINUED in the **srcsrpy** subroutine *Continued* parameter.

> **Note:** The STOP subsystem action does not allow any kind of continuation. However, all other action requests received by the subsystem from the SRC may be sent an informative message.

The reply packet is passed back to the client for further processing. Therefore, the packet must be agreed upon by the subsystem and the requester. One example of this type of continuation is a status request. When responding to subsystem status requests, specify STATCONTINUED in the **srcsrpy** *Continued* parameter. When status reporting has completed, or all subsystem-defined reply packets have been sent, specify END in the **srcsrpy** *Continued* parameter. The packet is then passed to the client to indicate the end of the reply.

Programming Subsystems to Return SRC Error Packets

Subsystems are required to return error packets for both SRC errors and non-SRC errors.

When returning an SRC error, the reply packet that the subsystem returns should be in the form of the **svrreply** structure of the **srcrep** structure, with the `objname` field filled in with the subsystem name, subserver type, or subserver object in error. If the NLS message associated with the SRC error number does not include any tokens, the error packet is returned in short form. This means the error packet contains the SRC error number only. However, if tokens are associated with the error number, standard NLS message text from the message catalog should be returned.

When returning a non-SRC error, the reply packet should be the `rtncode` field of the **svrreply** structure set to the SRC_SUBMSG constant and the `rtnmsg` field set to a subsystem-specific NLS message. The `rtnmsg` field is printed to the client's standard output.

Responding to Trace Requests

Support for the **traceson** and **tracesoff** commands is subsystem-dependent. If you choose to support these commands, trace actions can be specified for subsystems and subservers.

Subsystem trace requests will arrive in the following form: A subsystem trace request will have the **subreq** `action` field set to the TRACE constant and the **subreq** `object` field set to the SUBSYSTEM constant. The trace action uses `parm1` to indicate LONGTRACE or SHORTTRACE trace, and `parm2` to indicate TRACEON or TRACEOFF.

When the subsystem receives a trace subsystem packet with `parm1` set to SHORTTRACE and `parm2` set to TRACEON, the subsystem should turn short tracing on. Conversely, when the subsystem receives a trace subsystem packet with `parm1` set to LONGTRACE and `parm2` set to TRACEON, the subsystem should turn long tracing on. When the subsystem receives a trace subsystem packet with `parm2` set to TRACEOFF, the subsystem should turn subsystem tracing off.

Subserver trace requests will arrive in the following form: the subserver trace request will have the **subreq** `action` field set to the TRACE constant and the **subreq** `object` field set to the subserver code point of the subserver to send status on. The trace action uses `parm1` to indicate LONGTRACE or SHORTTRACE, and `parm2` to indicate TRACEON or TRACEOFF.

When the subsystem receives a trace subserver packet with `parm1` set to SHORTTRACE and `parm2` set to TRACEON, the subsystem should turn subserver short tracing on. Conversely, when the subsystem receives a trace subserver packet with `parm1` set to LONGTRACE and `parm2` set to TRACEON, the subsystem should turn subserver long tracing on. When the subsystem receives a trace subserver packet with `parm2` set to TRACEOFF, the subsystem should turn subserver tracing off.

Responding to Refresh Requests

Support for subsystem refresh requests is subsystem-dependent. Subsystem programmers that choose to support the **refresh** command should program their subsystems to interact with the SRC in the following manner:

- A subsystem refresh request will have the **subreq** structure `action` field set to the REFRESH constant and the **subreq** structure `object` field set to the SUBSYSTEM constant. The refresh subsystem action does not use `parm1` or `parm2`.
- When the subsystem receives the refresh request, the subsystem should reconfigure itself.

Defining Your Subsystem to the SRC

Subsystems are defined to the SRC object class as subsystem objects. Subservers are defined in the SRC configuration database as subserver type objects. The structures associated with each type of object are predefined in the **sys/srcobj.h** file.

A subsystem object is created with the **mkssys** command or the **addssys** subroutine. A subserver type object is created with the **mkserver** command. You are not required to specify all possible options and parameters using the configuration commands and subroutines. The SRC offers pre-set defaults. You must specify only the required fields and any fields in which you want some value other than the default. See the Subsystem Object Descriptor and Default Value table in “Subsystem Object Class” on page 699 in ″SRC Objects″ for a list of subsystem and subserver default values.

Descriptors can be added or modified at the command line by writing a shell script. They can also be added or modified using the C interface. Commands and subroutines are available for configuring and modifying the SRC objects.

> **Note:** The choice of programming interfaces is provided for convenience only.

At the command line use the following commands:

mkssys	Adds a subsystem definition to the SRC configuration database.
mkserver	Adds a subserver definition to the SRC configuration database.
chssys	Changes a subsystem definition in the SRC configuration database.
chserver	Changes a subserver definition in the SRC configuration database.
rmssys	Removes a subsystem definition from the SRC configuration database.
rmserver	Removes a subserver definition from the SRC configuration database.

When using the C interface, use the following subroutines:

addssys	Adds a subsystem definition to the SRC configuration database
chssys	Changes a subsystem definition in the SRC configuration database
defssys	Initializes a new subsystem definition with default values
delssys	Deletes an existing subsystem definition from the SRC configuration database **Note:** The object code running with the **chssys** subroutine must be running with the group system.
getssys	Gets a subsystem definition from the SRC configuration database
getsubsvr	Gets a subserver definition from the SRC configuration database

The **mkssys** and **mkserver** commands call the **defssys** subroutine internally to determine subsystem and subserver default values prior to adding or modifying any values entered at the command line.

The **getssys** and **getsubsvr** subroutines are used when the SRC master program or a subsystem program needs to retrieve data from the SRC configuration files.

List of Additional SRC Subroutines

Use the following subroutines to program communication with the SRC and the subsystems controlled by the SRC:

src_err_msg	Returns message text for SRC errors encountered by SRC library routines. (Also see threadsafe version **src_err_msg_r**)
srcsbuf	Requests status from the subsystem in printable format. (Also see threadsafe version **srcsbuf_r**)
srcsrqt	Sends a message or request to the subsystem. (Also see threadsafe version **srcsrqt_r**)
srcstat	Requests short subsystem status. (Also see threadsafe version **srcstat_r**)
srcstathdr	Gets the title text for SRC status.
srcstattxt	Gets the text representation for an SRC status code. (Also see threadsafe version **srcstattxt_r**)
srcstop	Requests termination of the subsystem.
srcstrt	Requests the start of a subsystem.

Chapter 27. Trace Facility

The trace facility helps you isolate system problems by monitoring selected system events. Events that can be monitored include: entry and exit to selected subroutines, kernel routines, kernel extension routines, and interrupt handlers. When the trace facility is active, information about these events is recorded in a system trace log file. The trace facility includes commands for activating and controlling traces and generating trace reports. Applications and kernel extensions can use several subroutines to record additional events.

The Trace Facility Overview

The trace facility is in the bos.sysmgt.trace file set. To see if this file set is installed, use

lslpp -l | grep bos.sysmgt.trace

If a line is produced which includes **bos.sysmgt.trace** then the file set is installed, otherwise you must install it.

The following topics are discussed:

- Overview
- Controlling the trace
- Recording Trace Event Data
- Generating a Trace Report
- Extracting trace data from a dump

The AIX system trace facility records trace events which can be formatted later by the trace report command. Trace events are compiled into kernel or application code, but are only traced if tracing is active.

Tracing is activated with the **trace** command or the **trcstart** subroutine. Tracing is stopped with either the **trcstop** command or the **trcstop** subroutine. While active, tracing can be suspended or resumed with the **trcoff** and **trcon** commands, or the **trcoff** and **trcon** subroutines.

Once the trace has been stopped with **trcstop**, a trace report can then be generated with the **trcrpt** command. This command uses a template file, **/etc/trcfmt**, to know how to format the entries. The templates are installed with the **trcupdate** command. For a discussion of the templates, see the trcupdate command.

Controlling the Trace

The **trace** command starts the tracing of system events and controls the trace buffer and log file sizes. This command is documented in the article on the trace daemon in the Command's Reference.

There are three methods of gathering trace data.

1. The default method is to use 2 buffers to continuously gather trace data, writing one buffer while data is being put into the other buffer. The log file wraps when it becomes full.
2. The circular method gathers trace data continuously, but only writes the data to the log file when the trace is stopped. This is particularly useful for debugging a problem where you know when the problem is happening and you just want to capture the data at that time. You can start the trace at any time, and then stop it right after the problem occurs and you'll have captured the events around the problem. This method is enabled with the **-l** trace daemon flag.
3. The third option only uses one trace buffer, and quits tracing when that buffer fills, and writes the buffer to the log file. The trace is not stopped at this point, rather tracing is turned off as if a **trcoff** command

had been issued. At this point you will usually want to stop the trace with the trcstop command. This option is most often used to gather performance data where we don't want trace to do i/o or buffer swapping until the data has been gathered. Use the **-f** flag to enable this option.

You will usually want to run the trace command asynchronously, in other words, you want to enter the trace command and then continue with other work. To run the trace asynchronously, use the **-a** flag. You must then stop the trace with the **trcstop** command.

It is usually desirable to limit the information that is traced. Use the **-j events** or **-k events** flags to specify a set of events to include (-j) or exclude (-k). Note, however, that to be able to display the program names associated with trace hooks, certain hooks must be enabled. These are given in the trace command article. These hooks may also be enabled by selecting the **tidhk** trace hook group from SMIT.

For example, if you want to trace the mbuf hook, 254, and show program names also, you need to run **trace** as follows:

trace -aj 106,10c,134,139,465,254

Tracing occurs.

trcstop

trcrpt -O exec=on

The hooks needed for showing the program name may change from release to release, so please check the **trace command** article. The **-O exec=on** trcrpt option shows the program names, see the trcrpt command.

It is often desirable to specify the buffer size and the maximum log file size. The trace buffers require real memory to be available so that no paging is necessary to record trace hooks. The log file will fill to the maximum size specified, and then wrap around, discarding the oldest trace data. The **-T size** and **-L size** flags specify the size of the memory buffers and the maximum size of the trace data in the log file in bytes.

Note: Because the trace facility pins the data collection buffers, making this amount of memory unavailable to the rest of the system, the trace facility can impact performance in a memory-constrained environment. If the application being monitored is not memory-constrained, or if the percentage of memory consumed by the trace routine is small compared to what is available in the system, the impact of trace "stolen" memory should be small. If you do not specify a value, trace uses the default sizes.

It should also be noted that the buffer memory is allocated from the kernel heap by default. If there isn't enough available memory, it is allocated from separate segments. On a busy system it may be desirable to allocate the buffers from separate memory segments to avoid constraining the kernel heap. This is done with the **-B** flag on the **trace** command.

Tracing may also be controlled from an application. See the trcstart, and trcstop articles.

Recording Trace Event Data

There are two types of trace data.

generic data
: consists of a data word, a buffer of opaque data and the opaque data's length. This is useful for tracing items such as path names. See the Generic Trace Channels article in the **Trace Facility Overview.** It can be found in "Chapter 27. Trace Facility" on page 713.

Non-generic data

This is what is normally traced by the AIX operating system. Each entry of this type consists of a hook word and up to 5 words of trace data. For a 64-bit application these are 8-byte words. The C programmer should use the macros TRCHKL0 through TRCHKL5, and TRCHKL0T through TRCHKL5T defined in the **/usr/include/sys/trcmacros.h** file, to record non-generic data. If these macros can not be used, see the article on the utrchook subroutine.

Generating a Trace Report

See the trcrpt command article for a full description of **trcrpt.** This command is used to generate a readable trace report from the log file generated by the **trace** command. By default the command formats data from the default log file, **/var/adm/ras/trcfile**. The **trcrpt** output is written to standard output.

To generate a trace report from the default log file, and write it to /tmp/rptout, enter

trcrpt >/tmp/rptout

To generate a trace report from the log file /tmp/tlog to /tmp/rptout, which includes program names and system call names, use

trcrpt -O exec=on,svc=on /tmp/tlog >/tmp/rptout

Extracting trace data from a dump

If trace was active when the system takes a dump, the trace can usually be retrieved with the trcdead command. To avoid overwriting the default trace log file on the current system, use the **-o output-file** option.

For example

trcdead /o /tmp/tlog /var/adm/ras/vmcore.0

creates a trace log file /tmp/tlog which may then be formatted with

trcrpt /tmp/tlog

Trace Facility Commands

The following commands are part of the trace facility:

trace	Starts the tracing of system events. With this command, you can control the size and manage the trace log file as well as the internal trace buffers that collect trace event data.
trcdead	Extracts trace information from a system dump. If the system halts while the trace facilities are active, the contents of the internal trace buffers are captured. This command extracts the trace event data from the dump and writes it to the trace log file.
trcnm	Generates a kernel name list used by the **trcrpt** command. A kernel name list is composed of a symbol table and a loader symbol table of an object file. The **trcrpt** command ues the kernel name list file to inerpret addresses when formatting a report from a trace log file.

trcrpt	Formats reports of trace event data contained in the trace log file. You can specify the events to be included (or omitted) in the report, as well as determine the presentation of the output with this command. The **trcrpt** command uses the trace formatting templates stored in the **/etc/trcfmt** file to determine how to interpret the data recorded for each event.
trcstop	Stops the tracing of system events.
trcupdate	Updates the trace formatting templates stored in the **/etc/trcfmt** file. When you add applications or kernel extensions that record trace events, templates for these events must be added to the **/etc/trcfmt** file. The **trcrpt** command will use the trace formatting templates to determine how to interpret the data recorded for each event. Software products that record events usually run the **trcupdate** command as part of the installation process.

Trace Facility Calls and Subroutines

The following calls and subroutines are part of the trace facility:

trcgen, **trcgent**	Records trace events of more than five words of data. The **trcgen** subroutine may be used to record an event as part of the system event trace (trace channel 0) or to record an event on a generic trace channel (channels 1 through 7). You specify the channel number in a subroutine parameter when you record the trace event. The **trcgent** subroutine appends a time stamp to the event data.
trchook, **utrchook**	Records trace events of up to five words of data. These subroutines may be used to record an event as part of the system event trace (trace channel 0). The **utrchook** subroutine uses a special FAST-SVC path to improve performance and should be used by programs at the user (application) level.
trcgenk, trcgenkt	Records trace events of more than five words of data. The **trcgenk** subroutine may be used to record an event as part of the system event trace (trace channel 0) or to record an event on a generic trace channel (channels 1 through 7). You specify the channel number in a subroutine parameter when you record the trace event. The **trcgenkt** subroutine appends a time stamp to the event data.
trcoff	Suspends the collection of trace data on either the system event trace channel (channel 0) or a generic trace channel (1 through 7). The trace channel remains active and trace data collection can be resumed by using the **trcon** subroutine.
trcon	Starts the collection of trace data on a trace channel. The channel may be either the system event trace channel (0) or a generic channel (1 through 7). The trace channel, however, must have been previously activated by using the **trace** command or the **trcstart** subroutine. You can suspend trace data collection by using the **trcoff** subroutine.

trcstart	Requests a generic trace channel. This subroutine activates a generic trace channel and returns the channel number to the calling application to use in recording trace events using the **trcgen**, **trcgent**, **trcgenk**, and **trcgenkt** subroutines.
trcstop	Frees and deactivates a generic trace channel.

Trace Facility Files

/etc/trcfmt	Contains the trace formatting templates used by the **trcrpt** command to determine how to interpret the data recorded for each event.
/var/adm/ras/trcfile	Contains the default trace log file. The **trace** command allows you to specify a different trace log file.
/usr/include/sys/trchkid.h	Contains trace hook identifier definitions.
/usr/include/sys/trcmacros.h	Contains commonly used macros for recording trace events.

Trace Event Data

The data recorded for each traced event consist of a word containing the trace hook identifier and the hook type followed by a variable number of words of trace data optionally followed by a time stamp. The word containing the trace hook identifier and the hook type is called the hook word. The remaining two bytes of the hook word are called hook data and are available for recording event data.

Trace Hook Identifiers

A trace hook identifier is a three-digit hexadecimal number that identifies an event being traced. You specify the trace hook identifier in the first twelve bits of the hook word. Trace hook identifiers are defined in the **/usr/include/sys/trchkid.h** file. The values 0x010 through 0x0FF are available for use by user applications. All other values are reserved for system use. The currently defined trace hook identifiers can be listed using the **trcrpt -j** command.

Hook Types

The hook type identifies the composition of the event data and is user-specified. The twelfth through the sixteenth bits of the hook word constitute the hook type. For more information on hook types, refer to the **trcgen**, **trcgenk**, and **trchook** subroutines.

Trace Facility Generic Trace Channels

The trace facility supports up to eight active trace sessions at a time. Each trace session uses a channel of the multiplexed trace special file, **/dev/systrace**. Channel 0 is used by the trace facility to record system events. The tracing of system events is started and stopped by the **trace** and **trcstop** commands. Channels 1 through 7 are referred to as generic trace channels and may be used by subsystems for other types of tracing such as data link tracing.

To implement tracing using the generic trace channels of the trace facility, a subsystem calls the **trcstart** subroutine to activate a trace channel and to determine the channel number. The subsystem modules can then record trace events using the **trcgen**, **trcgent**, **trcgenk**, or **trcgenkt** subroutine. The channel number returned by the **trcstart** subroutine is one of the parameters that must be passed to these subroutines. The subsystem can suspend and resume trace data collection using the **trcoff** and **trcon** subroutines and can deactivate a trace channel using the **trcstop** subroutine. The trace events for each channel must be written to a separate trace log file, which must be specified in the call to the **trcstart** subroutine. The subsystem must provide the user interface to activating and deactivating subsystem tracing.

The trace hook IDs, which are stored in the **/usr/include/sys/trchkid.h** file, and the trace formatting templates, which are stored in the **/etc/trcfmt** file, are shared by all the trace channels.

Related Information

The **trace** daemon in *AIX 5L Version 5.1 Commands Reference*.

The **trcdead** command, **trcnm** command, **trcrpt** command, **trcstop** command, **trcupdate** command in *AIX 5L Version 5.1 Commands Reference*.

The **trchook** subroutine, **trcgen** subroutine, **trcoff** subroutine, **trcon** subroutine, **trcstart** subroutine, **trcstop** subroutine in *AIX 5L Version 5.1 Technical Reference: Base Operating System and Extensions Volume 2*.

Start the Trace Facility

Use the following procedures to configure and start a system trace:

Configuring the trace Command

The **trace** command starts the tracing of system events and controls the size of and manages the **trace** log file, as well as the internal trace buffers that collect trace event data. The syntax of this command is:

```
trace [-fl] [-ad] [-s] [-h] [-jk events] [,events] [-m message] [-o outfile][-g] [-T buf_sz] [-L log_sz]
```

The various options of the **trace** command are:

-f or -l
: Controls the capture of trace data in system memory. If you specify neither the **-f** nor **-l** option, the trace facility creates two buffer areas in system memory to capture the trace data. The **trace** log files and the internal **trace** buffers that collect trace event data can be managed, including their size, by this command. The **-f** or **-l** flag provides the ability to prevent data from being written to the file during data collection. The options are to collect data only until the memory buffer becomes full (**-f** for first), or to use the memory buffer as a circular buffer that captures only the last set of events that occurred before **trace** was terminated (**-l**). The **-f** and **-l** options are mutually exclusive. With either the **-f** or **-l** option, data is not transferred from the memory collection buffers to file until **trace** is terminated.

-a
: Runs the **trace** collection asynchronously (as a background task), returning a normal command line prompt. Without this option, the **trace** command runs in a subcommand mode and returns a > prompt. You can issue subcommands and regular shell commands from the **trace** subcommand mode by preceding the shell commands with an ! (exclamation point).

-d
: Delays data collection. The trace facility is only configured. Data collection is delayed until one of the collection trigger events occurs. Various methods for triggering data collection on and off are provided. These include the following:
  - **trace** subcommands
  - **trace** commands
  - **trace** subroutines.

-j events or -k events
Specifies a set of events to include (**-j**) or exclude (**-k**) from the collection process. Specifies a list of events to include or exclude by a series of three-digit hexadecimal event IDs separated by a space.

-s
Terminate trace data collection if the **trace** log file reaches its maximum specified size. The default without this option is to wrap and overwrite the data in the log file on a FIFO basis.

-h
Does not write a **date/sysname/message** header to the **trace** log file.

-m message
Specifies a text string (message) to be included in the **trace** log header record. The message is included in reports generated by the **trcrpt** command.

-o outfile
Specifies a file to use as the log file. If you do not use the **-o** option, the default log file is **/var/adm/ras/trcfile**. To direct the trace data to standard output, code the **-o** option as **-o -**. Use this technique only to pipe the data stream to another process since the trace data contains raw binary events that are not displayable.

-g
Duplicates the **trace** design for multiple channels. Channel 0 is the default channel and is always used for recording system events. The other channels are generic channels, and their use is not predefined. There are various uses of generic channels in the system. The generic channels are also available to user applications. Each created channel is a separate events data stream. Events recorded to channel 0 are mixed with the predefined system events data stream. The other channels have no predefined use and are assigned generically.

A program typically requests that a generic channel be opened by using the **trcstart** subroutine. A channel number is returned, similar to the way a file descriptor is returned when a file is opened (the channel ID). The program can record events to this channel and, thus, have a private data stream. Less frequently, the **trace** command allows a generic channel to be specifically configured by defining the channel number with this option.

-T size and -L size
Specifies the size of the collection memory buffers and the maximum size of the log file in bytes.

> **Note:** Because the trace facility pins the data collection buffers, making this amount of memory unavailable to the rest of the system, the trace facility can impact performance in a memory-constrained environment. If the application being monitored is not memory-constrained, or if the percentage of memory consumed by the **trace** routine is small compared to what is available in the system, the impact of **trace** ″stolen″ memory should be small.

If you do not specify a value, trace uses a default size.

Recording Trace Event Data

The data recorded for each traced event consist of a word containing the trace hook identifier and the hook type followed by a variable number of words of trace data optionally followed by a time stamp. The word containing the trace hook identifier and the hook type is called the hook word. The remaining two bytes of the hook word are called hook data and are available for recording event data.

Trace Hook Identifiers

A trace hook identifier is a three-digit hexadecimal number that identifies an event being traced. You specify the trace hook identifier in the first 12 bits of the hook word. The values 0x010 through 0x0FF are available for use by user applications. All other values are reserved for system use. The trace hook identifiers for the installed software can be listed using the **trcrpt -j** command.

The trace hook IDs, which are stored in the **/usr/include/sys/trchkid.h** file, and the trace formatting templates, which are stored in the **/etc/trcfmt** file, are shared by all the trace channels.

Hook Types

The hook type identifies the composition of the event data and is user-specified. Bits 12 through 16 of the hook word constitute the hook type. For more information on hook types, refer to the **trcgen**, **trcgenk**, and **trchook** subroutines.

Using Generic Trace Channels

The trace facility supports up to eight active trace sessions at a time. Each trace session uses a channel of the multiplexed trace special file, **/dev/systrace**. Channel 0 is used by the trace facility to record system events. The tracing of system events is started and stopped by the **trace** and **trcstop** commands. Channels 1 through 7 are referred to as generic trace channels and may be used by subsystems for other types of tracing such as data link tracing.

To implement tracing using the generic trace channels of the trace facility, a subsystem calls the **trcstart** subroutine to activate a trace channel and to determine the channel number. The subsystem modules can then record trace events using the **trcgen**, **trcgent**, **trcgenk**, or **trcgenkt** subroutine. The channel number returned by the **trcstart** subroutine is one of the parameters that must be passed to these subroutines. The subsystem can suspend and resume trace data collection using the **trcoff** and **trcon** subroutines and can deactivate a trace channel using the **trcstop** subroutine. The trace events for each channel must be written to a separate **trace** log file, which must be specified in the call to the **trcstart** subroutine. The subsystem must provide the user interface for activating and deactivating subsystem tracing.

Starting a Trace

Use the one of the following procedures to start the trace facility.

- Start the trace facility by using the **trace** command.

  Start the trace asynchronously. For example:

  ```
  trace -a
  mycmd
  trcstop
  ```

  When using the trace facility asynchronously, use the **trace** daemon to trace the selected system events (such as the **mycmd** command); then, use the **trcstop** command to stop the trace.

  OR

  Start the trace interactively. For example:

  ```
  trace
  ->!mycmd
  ->quit
  ```

  When using the trace facility interactively, get into the interactive mode as denoted by the -> prompt, and use the **trace** subcommands (such as **!**) to trace the selected system events. Use the **quit** subcommand to stop the trace.
- Use **smit trace**, and choose the **Start Trace** option.

  ```
  smit trace
  ```

Stopping a Trace

Use one of the following procedures to stop the trace you started earlier.

- When using **trace** asynchronously at the command line, use the **trcstop** command:

```
trace -a
mycmd
trcstop
```

When using the trace facility asynchronously, use the **trace** daemon to trace the selected system events (such as the **mycmd** command); then, use the **trcstop** command to stop the trace.

- When using **trace** interactively at the command line, use the **quit** subcommand:

```
trace
->!mycmd
->quit
```

The interactive mode is denoted by the -> prompt. Use the **trace** subcommands (such as **!**) to trace the selected system events. Use the **quit** subcommand to stop the trace.

- Use **smit trace** and choose the **Stop Trace** option:

```
smit trace
```

Generating a Trace Report

Use either of the following procedures to generate a report of events that have been traced.

- Use the **trcrpt** command:

```
trcrpt>/tmp/NewFile
```

The previous example formats the **trace** log file and sends the report to **/tmp/newfile**. The **trcrpt** command reads the **trace** log file, formats the trace entries, and writes a report.

- Use the **smit trcrpt** command:

```
smit trcrpt
```

Trace Hook IDs: 001 through 10A

The following trace hook IDs are stored in the **/usr/include/sys/trchkid.h** file.

001 : HKWD TRACE TRCON

This event is recorded by the **trcon** ioctl of the **/dev/systrcctl** file.

Recorded Data

TRACE ON channel *channel number*

channel *channel number* Trace channel number:

0 System event trace
1-7 Generic trace channels.

002 : HKWD TRACE TRCOFF

This event is recorded by the **trcoff** ioctl of the **/dev/systrcctl** file.

Recorded Data

TRACE OFF channel *channel number*

channel *channel number* Trace channel number:

0 System event trace
1-7 Generic trace channels.

003 : HKWD TRACE HEADER

This event is used to record the timestamp and the system information that appear at the top of the trace report.

Recorded Data

timestamp **System** *system name* **Machine** *machine id* **Internet Address** *internet address*

timestamp	Date and time the trace log was created
System *system name*	Operating system name, release, and version
Machine *machine id*	The machine ID
Internet Address *internet address*	The Internet address of this machine.

004 : HKWD TRACE NULL

This hook ID is used to provide a template for formatting events for which the trace hook ID is 000.

Recorded Data

TRACEID IS ZERO hookword=*hookword* **file=***file name* **index=***value*

hookword=*hookword*	The contents of the hook word
file=*file name*	The trace log file pathname
index=*value*	The offset into the trace log file of the event.

005 : HKWD TRACE LWRAP

The **trace** daemon records this hook ID each time the trace log file wraps.

Recorded Data

LOGFILE WRAPAROUND *count*

Wraparound *count*	Number of times log file has wrapped.

006 : HKWD TRACE TWRAP

This event is recorded by the **trchk** and **trcgen** subroutines each time the trace buffer wraps.

Recorded Data

TRACEBUFFER Wraparound *count value* **missed entries**

Wraparound *count*	Number of times trace buffer has wrapped
value **missed entries**	Number of entries overwritten.

007 : HKWD TRACE UNDEFINED

This hook ID is used to provide a template for formatting undefined events. Events in the trace log file for which there is no template defined in the **/etc/trcfmt** file are formatted using this template.

Recorded Data

UNDEFINED TRACE ID idx *offset* **traceid** *trace id* **hookword** *hookword* **type** *hook type* **hookdata** *data*

idx *offset*	Offset of event into the trace log file
traceid *trace id*	Trace hook ID of undefined event
hookword *hookword*	The contents of the hook word for the event
type *hook type*	The hook type (0-7)
hookdata *data*	The data recorded for the event is printed in hexadecimal.

100 : HKWD KERN FLIH

This event is recorded by the First Level Interrupt Handler in the event of a first-level interrupt. Return from FLIH is recorded by hook ID **200 : HKWD KERN RESUME**.

Recorded Data

Type of interrupt:

Machine Check

Data Access Page Fault

Instruction Page Fault

I/O Interrupt

Alignment Error

Program Check

Floating Point Unavailable

101 : HKWD KERN SVC

This event is recorded by SVC handler on entry to a subroutine.

Recorded Data

Name of the subroutine.

102 : HKWD KERN SLIH

This event is recorded by the Second Level Interrupt Handler in the event of a second-level interrupt. Return from SLIH is recorded by hook ID **103 : HKWD KERN SLIHRET**.

Recorded Data

The name of the SLIH function.

103 : HKWD KERN SLIHRET

This event ID is recorded by the Second Level Interrupt Handler on return from a second-level interrupt.

Recorded Data

return from slih

104 : HKWD KERN SYSCRET

This event is recorded by the SVC handler on return from a subroutine.

Recorded Data

return from *subroutine* **error** *errno*

subroutine	Name of the subroutine
error *errno*	If *errno* is nonzero, the value of the **errno** global variable is printed.

105 : HKWD KERN LVM

This event is recorded by the Logical Volume Manager for selected events.

Recorded Data

Event:

LVM relocingblk bp=*value* **pblock=***value* **relblock=***value* Encountered relocated block

bp=*value*
Buffer pointer

pblock=*value*
Physical block number

relblock=*value*
Relocated block number.

LVM oldbadblk bp=*value* **pblock=***value* **state=***value* *bflags* Bad block waiting to be relocated

bp=*value*	Buffer pointer
pblock=*value*	Physical block number
state=*value*	State of the physical volume
bflags	Buffer flags are defined in the **sys/buf.h** file.

LVM badblkdone bp=*value* Block relocation complete

bp=*value* Buffer pointer.

LVM newbadblk bp=*value* **badblock=***value* **error=***value* *bflags* New bad block found

bp=*value*	Buffer pointer
badblock=*value*	Block number of bad block
error=*value*	System error number (the **errno** global variable)
bflags	Buffer flags are defined in the **sys/buf.h** file.

LVM swreloc bp=*value* **status=***value* **error=***value* **retry=***value* Software relocating bad block

bp=*value*	Buffer pointer
status=*value*	Bad block directory entry status

error=*value* System error number (the **errno** global variable)
retry=*value* Relocation entry count.

LVM resyncpp bp=*value bflags* Resyncing Logical Partition mirrors

bp=*value* Buffer pointer
bflags Buffer flags are defined in the **sys/buf.h** file.

LVM open *device name* **flags=***value* Open

device name Name of the device
flags=*value* Open file mode.

LVM close *device name* Close

device name Name of the device.

LVM read *device name* **ext=***value* Read

device name Name of the device
ext=*value* Extension parameters.

LVM write *device name* **ext=***value* Write

device name Name of the device
ext=*value* Extension parameters.

LVM ioctl *device name* **cmd=***value* **arg=***value* ioctl

device name Name of the device
cmd=*value* **ioctl** command
arg=*value* **ioctl** arguments.

106 : HKWD KERN DISPATCH

This event is recorded by the dispatcher when a process thread is dispatched.

Recorded Data

dispatch *process name process id*

process name Name of the dispatched process
process id Process ID of the dispatched process.

dispatch cmd=*process name* **pid=***process id* **tid=***thread id* **priority=***priority* **old_tid=***old thread id* **old_priority=***old priority*
dispatch scheduler
process name Process name of the dispatched thread.

process id	Process ID of the dispatched thread.
thread id	Thread ID of the dispatched thread.
priority	Priority of the dispatched thread.
old thread id	Thread ID of the thread that dispatches.
old priority	Priority of the thread that dispatches.

107 : HKWD LFS LOOKUP

This event is recorded by the **lookuppn** kernel service.

Recorded Data

lookuppn *pathname*

pathname	Path name of the current file.

108 : HKWD SYSC LFS

This event is recorded by the file system related subroutines.

Recorded Data

Event:

access *file mode*	**access** subroutine
fchmod *file mode*	**fchmod** subroutine
chown *file name* **uid=***value* **gid=***value*	**chown** subroutine
fchown *file name* **uid=***value* **gid=***value*	**fchown** subroutine
chownx *file name* **uid=***value* **gid=***value*	**chownx** subroutine
fchownx *file name* **uid=***value* **gid=***value*	**fchownx** subroutine
ftruncate *file name* **to** *length*	**ftruncate** subroutine
truncate *file name* **to** *length*	**truncate** subroutine
ioctlx *file name* **cmd=***value*	**ioctlx** subroutine
lockfx *file name* **start=***value* **length=***value* **whence=***value*	**lockfx** subroutine
mknod *file name file mode*	**mknod** subroutine
fsync *file name*	**fsync** subroutine
readx (*fd,buf,count***)** *file name*	**readx** subroutine
writex (*fd,buf,count***)** *file name*	**writex** subroutine

openx *file name* **fd=***value file mode*

openx subroutine

file name
File path name

uid=*value*
User ID

gid=*value*
Group ID

fd=*value*
File descriptor

file mode
File mode

to *length*
Length to truncate to

cmd=*value*
ioctl operation

start=*value*
Starting offset

length=*value*
Length to lock

whence=*value*
Type of lock

(*fd,buf,count***)**
File descriptor, buffer pointer, and count.

10A : HKWD KERN PFS

This event is recorded by the kernel physical file system for selected events.

Recorded Data

Event:

PFS rdwr (vp, ip)=(*vp, ip***)** *filename*

PFS readi VA.S=*value* **bcount=***value* **ip=***value filename*

PFS writei VA.S=*value* **bcount=***value* **ip=***value filename*

(vp, ip)=(*vp, ip***)**

vp	v_node pointer
ip	i_node pointer
filename	File path name
VA.S=*value*	Segment ID that maps the file
bcount=*value*	Byte count.

Trace Hook IDs: 10B through 14E

The following trace hook IDs are stored in the **/usr/include/sys/trchkid.h** file.

10B : HKWD KERN LVMSIMP

This event is recorded by Logical Volume Manager for selected events.

Recorded Data

Event:

LVM rblocked: bp=*value* Request blocked by conflict resolution

bp=*value*
Buffer pointer.

LVM pend: bp=*value* **resid=***value* **error=***value bflags* End of physical operation

bp=*value* Buffer pointer
resid=*value* Residual byte count
error=*value* System error number (the **errno** global variable)
bflags Buffer flags are defined in the **sys/buf.h** file.

bp=*value* Buffer pointer
resid=*value* Residual byte count
error=*value* System error number (the **errno** global variable)
bflags Buffer flags are defined in the **sys/buf.h** file.

LVM lstart: *device name* **bp=***value* **lblock=***value* **bcount=***value bflags* **opts:***value* Start of logical operation

device name Device name
bp=*value* Buffer pointer
lblock=*value* Logical block number
bcount=*value* Byte count
bflags Buffer flags are defined in the **sys/buf.h** file
opts: *value* Possible values:

WRITEV

HWRELOC

UNSAFEREL

RORELOC

NO_MNC

MWC_RCV_OP

RESYNC_OP

AVOID_C1

AVOID_C2

AVOID_C3

device name
Device name

pblock=*value* Physical block number

(lbp,pbp)=(*lbp,pbp***)**	Description of variables: *lbp* Logical buffer pointer *pbp* Physical buffer pointer.
opts: *value*	Possible values: **WRITEV** **HWRELOC** **UNSAFEREL** **RORELOC** **NO_MNC** **MWC_RCV_OP** **RESYNC_OP** **AVOID_C1** **AVOID_C2** **AVOID_C3**

bflags Buffer flags are defined in the **sys/buf.h** file

filename File path name.

10C : HKWD KERN IDLE

This event is recorded by the dispatcher when dispatching a thread of the idle process.

Recorded Data

dispatch: idle process pid=*process id* **tid=***thread id* **priority=***priority* **old_tid=***old thread id* **old_priority=***old priority*

process id	Process ID of the dispatched thread.
thread id	Thread ID of the dispatched thread.
priority	Priority of the dispatched thread.
old thread id	Thread ID of the thread that dispatches.
old priority	Priority of the thread that dispatches.

10F : HKWD KERN EOF

This event is recorded by the kernel end of a file routine.

Recorded Data

KERN_EOF hookdata *data*

hookdata *data*	The data printed for this event is recorded in hexadecimal.

110 : HKWD KERN STDERR

This event is recorded by the kernel **stderr** routine.

Recorded Data

KERN_STERR hookdata *data*

hookdata *data*	The data recorded for the event is printed in hexadecimal.

112 : HKWD KERN LOCK

This event is recorded on each lock request.

Recorded Data

lock: *sub-hook* **lock addr=***lock* **lock status=***content* **request_mode=***mode* **return addr=***address* **name=***name*

sub-hook	Possible values: **lock** **miss** **recu** **busy**
lock	Address of the lock.
content	Content of the lock

Possible values:

LOCK_WRITE

LOCK_READ

LOCK_UPGRADE

LOCK_DOWNGRADE

address - Return address of the call.

name

113 : HKWD KERN UNLOCK

This event is recorded on each unlock request.

Recorded Data

unlock: lock addr=*lock* **lock status=***content* **return addr=***address* **name=***name*

lock	Address of the lock.
content	Content of the lock
address	Return address of the call.

name

114 : HKWD KERN LOCKALLOC

This event is recorded when allocating a lock.

Recorded Data

lockalloc: lock addr=*lock* **name=***class.occurence* **return addr=***address*

lock	Address of the lock.
class	Class name of the lock.
occurence	Index of the lock in the class.
address	Return address of the call.

115 : HKWD KERN SETRECURSIVE

This event is recorded by the **lock_set_recursive** and **lock_clear_recursive** kernel services.

Recorded Data

SETRECURSIVE lock addr=*lock* **return addr=***address*

CLEARRECURSIVE lock addr=*lock* **return addr=***address*

lock	Address of the lock.
address	Return address of the call.

116 : HKWD KERN XMALLOC

This event is recorded by the kernel **xmalloc** routine.

Recorded Data

xmalloc (*size*, *align*, *heap***)**

size	Number of bytes to allocate
align	Alignment characteristics for the allocated memory
heap	Address of the heap from which memory is to be allocated.

117 : HKWD KERN XMFREE

This event is recorded by the kernel **xmfree** routine.

Recorded Data

xfree (*address, heap***)**

address	Address of area in memory to free
heap	Address of the heap from which memory is to be allocated.

118 : HKWD KERN FORKCOPY

This event is recorded by the **forkcopy** routine.

Recorded Data

vmm_forkcopy

119 : HKWD KERN SENDSIGNAL

This event is recorded by the kernel **sendsignal** routine.

Recorded Data

KERN_SENDSIGNAL hookdata *data*

hookdata *data* The data recorded for this event is printed in hexadecimal.

11A : HKWD KERN RCVSIGNAL

This event is recorded by the kernel **rcvsignal** routine.

Recorded Data

KERN_RCVSIGNAL hookdata *data*

hookdata *data* The data recorded for this event is printed in hexadecimal.

11B : HKWD KERN LOCKL

This event is recorded by the kernel **lockl** routine.

Recorded Data

KERN_LOCKL hookdata *data*

hookdata *data* The data recorded for this event is printed in hexadecimal.

11C : HKWD KERN P SLIH

This event is recorded by the **sigreturn** routine.

Recorded Data

KERN_SIGRETURN hookdata *data*

hookdata *data* The data recorded for this event is printed in hexadecimal.

11D : HKWD KERN SIG SLIH

This event is recorded by the **sigdeliver** routine.

Recorded Data

KERN_SIGDELIVER hookdata *data*

hookdata *data* The data recorded for this event is printed in hexadecimal.

11E : HKWD KERN ISSIG

This event is recorded by the kernel **issig** routine.

Recorded Data

issig

11F : HKWD KERN SORQ

This event is recorded by the kernel set on ready queue routine.

Recorded Data

setrq: cmd=*process name* **pid=***process id* **tid=***thread id* **priority=***priority* **policy=***policy*

process name	Process name of the thread set on the ready queue.
process id	Process ID of the thread set on the ready queue.
thread id	Thread ID of the thread set on the ready queue.
priority	Priority of the thread set on the ready queue.
policy	Scheduling policy of the thread set on the ready queue.

120 : HKWD SYSC ACCESS

This event is recorded by the **access** subroutine.

Recorded Data

access mode=*value*

mode=*value*	Requested access.

121 : HKWD SYSC ACCT

This event is recorded by the **acct** subroutine.

Recorded Data

acct fname=*value*

fname=*value*	File path name.

122 : HKWD SYSC ALARM

This event is recorded by the **alarm** subroutine.

Recorded Data

alarm *secs* **seconds**

alarm off (zero seconds specified**)**

secs **seconds**	Number of seconds specified.

12E : HKWD SYSC CLOSE

This event is recorded by the **close** subroutine.

Recorded Data

close *filename* **fd=***value*

filename	File path name
fd=*value*	File descriptor.

134 : HKWD SYSC EXECVE

This event is recorded by the **exec** subroutine.

Recorded Data

File path name.

filename	File path name.
process id	Process ID.
thread id	Thread ID.

135 : HKWD SYSC EXIT

This event is recorded by the **exit** subroutine.

Recorded Data

exit wait_status=*value* **lockct=***value*

wait_status=value	Wait status
lockct=*value*	Lock count.

139 : HKWD SYSC FORK

This event is recorded by the **fork** subroutine.

Recorded Data

Process ID.

process id	Process ID.
thread id	Thread ID.

145 : HKWD SYSC GETPGRP

This event is recorded by the **getpgrp** subroutine.

Recorded Data

GETPGRP

146 : HKWD SYSC GETPID

This event is recorded by the **getpid** subroutine.

Recorded Data

GETPID

147 : HKWD SYSC GETPPID

This event is recorded by the **getppid** subroutine.

Recorded Data

GETPPID

14C : HKWD SYSC IOCTL

This event is recorded by the **ioctl** subroutine.

Recorded Data

Event:

ioctl fd=*value* **command=***value* **arg=***value*

ioctl fd=*value* **TCGETA**

ioctl fd=*value* **TCSETA**

ioctl fd=*value* **TCSETAW**

ioctl fd=*value* **TCSETAF**

ioctl fd=*value* **TCSBRK arg=***value*

ioctl fd=*value* **TCXONC arg=***value*

ioctl fd=*value* **TCXFLSH arg=***value*

fd=*value* File descriptor.

command=*value*

arg=*value*

14E : HKWD SYSC KILL

This event is recorded by the **kill** subroutine.

Recorded Data

signal *value* Signal name.
to process *process id*
process name

Trace Hook IDs: 152 through 19C

The following trace hook IDs are stored in the **/usr/include/sys/trchkid.h** file.

152 : HKWD SYSC LOCKF

This event is recorded by the **lockf** subroutine.

Recorded Data

Event:

lockf *filename* **fd=***value* **unlock** *value* **bytes**
lockf *filename* **fd=***value* **lock_wait** *value* **bytes**
lockf *filename* **fd=***value* **lock_busy** *value* **bytes**

filename	File path name
fd=*value*	File descriptor
value **bytes**	Number of bytes.

154 : HKWD SYSC LSEEK

This event is recorded by the **lseek** subroutine.

Recorded Data

Event:

lseek fd=*file descriptor* **to** *offset*
lseek fd=*file descriptor* **relative** *offset*
lseek fd=*file descriptor* **relative** *offset* **from end of file**
lseek fd=*file descriptor* **offset=***offset* **whence=***whence* **(whence)**

fd=*file descriptor*	File descriptor
offset=*offset*	Offset into file
relative *offset*	Offset into file
whence=*whence*	

Value	Meaning
0	From beginning
1	From current offset
2	From end of file.

15F : HKWD SYSC PIPE

This event is recorded by the **pipe** subroutine.

Recorded Data

pipe read_fd=*value* **write_fd=***value*

read_fd=*value*	Read file descriptor
write_fd=*value*	Write file descriptor.

160 : HKWD SYSC PLOCK

This event is recorded by the **pblock** subroutine.

Recorded Data

Event:

pblock *process* **UNLOCK**
pblock *process* **PROCESS LOCK**
pblock *process* **TEXT SEGMENT LOCK**
pblock *process* **DATA SEGMENT DATLOCK**

process
Process name.

169 : HKWD SYSC SBREAK

This event is recorded by the **sbreak** subroutine.

Recorded Data

sbreak new dmax is *value*

new dmax is *value*
Value of **dmax**.

16E : HKWD SYSC SETPGRP

This event is recorded by the **setpgid** subroutine.

Recorded Data

setpgid pid=*value* **pgrp=***value*
pid=*value* Process ID
pgrp=*value* Process group.

16F : HKWD SYSC SETPRIO

This event is recorded by the **sbreak** subroutine.

Recorded Data

SBREAK SUBROUTINE hookdata *data*

hookdata *data*
The data recorded for this event is printed in hexadecimal.

180 : HKWD SYSC SIGACTION

This event is recorded by the **sigaction** subroutine.

Recorded Data

sigaction signal *value* **mask=***value*

signal *value*
Signal number and name

mask=*value*
sigaction mask.

181 : HKWD SYSC SIGCLEANUP

This event is recorded by the **sigcleanup** subroutine.

Recorded Data

SIGCLEANUP

18E : HKWD SYSC TIMES

This event is recorded by the **times** subroutine.

Recorded Data

TIMES subroutine times u=*value* **s=***value* **cu=***value* **cs=***value* **(ticks)**

u=*value*
The CPU time (in ticks) used while executing instructions in the user space of the calling process

s=*value*
The CPU time (in ticks) used by the system on behalf of the calling process

cu=*value*
The CPU time (in ticks) used while executing instructions in the user space of child processes of the calling process

cs=*value*
The CPU time (in ticks) used by the system on behalf of child processes of the calling processes.

18F : HKWD SYSC ULIMIT

This event is recorded by the **ulimit** subroutine.

Recorded Data

Event:

ulimit get fsize
ulimit set fsize to *newlimit*
ulimit get data limit
ulimit set data limit to *newlimit*
ulimit get stack
ulimit set stack limit to *newlimit*
ulimit get RAWDIR compatibility mode (REALDIR)
ulimit clear RAWDIR compatibility mode (REALDIR)

ulimit set RAWDIR compatibility mode (REALDIR)
ulimit get TRUNCATE compatibility mode (SYSVLOOKUP)
ulimit clear TRUNCATE compatibility mode (SYSVLOOKUP)
ulimit set TRUNCATE compatibility mode (SYSVLOOKUP)

195 : HKWD SYSC USRINFO

This event is recorded by the **usrinfo** subroutine.

Recorded Data

usrinfo

19B : HKWD SYSC WAIT

This event is recorded by the **wait** subroutine.

Recorded Data

wait rv=*value* **pflag=***value* **wstat=***value*

rv=*value*
Value of the **rv** argument

pflag=*value*
Wait operation

wstat=*value*
Returned status.

Trace Hook IDs: 1A4 through 1BF

The following trace hook IDs are stored in the **/usr/include/sys/trchkid.h** file.

1A4 : HKWD SYSC GETRLIMIT

This event is recorded by the **getrlimit** subroutine.

Recorded Data

Event:

getrlimit resource=0 CPU TIME
getrlimit resource=1 MAX FILE SIZE
getrlimit resource=2 DATA SEGMENT SIZE
getrlimit resource=3 SIZE SIZE
getrlimit resource=4 CORE FILE SIZE
getrlimit resource=5 RESIDENT SET SIZE

1A5 : HKWD SYSC SETRLIMIT

This event is recorded by the **setrlimit** subroutine.

Recorded Data

Event:

setrlimit resource=0 CPU TIME
setrlimit resource=1 MAX FILE SIZE
setrlimit resource=2 DATA SEGMENT SIZE
setrlimit resource=3 SIZE SIZE
setrlimit resource=4 CORE FILE SIZE
setrlimit resource=5 RESIDENT SET SIZE

1A6 : HKWD SYSC GETRUSAGE

This event is recorded by the **getrusage** subroutine.

Recorded Data

Event:

getrusage who=*value* **of self**
getrusage who=*value* **of children**

who=*value* Possible values:

RUSAGE_SELF

RUSAGE_CHILDREN

1A7 : HKWD SYSC GETPRIORITY

This event is recorded by the **getpriority** subroutine.

Recorded Data

Event:

getpriority of process *process id process name*

getpriority of process group *process id process name*

getpriority of uid (current process)

1A8 : HKWD SYSC SETPRIORITY

This event is recorded by the **setpriority** subroutine.

Recorded Data

Event:

setpriority of process *process id process name*
setpriority of process group *process id process name*
setpriority of uid (current process)

1A9 : HKWD SYSC ABSINTERVAL

This event is recorded by the **absinterval** subroutine.

Recorded Data

absinterval timerid=*value*

timerid=*value*
Timer identifier.

1AA : HKWD SYSC GETINTERVAL

This event is recorded by the **getinterval** subroutine.

Recorded Data

getinterval timerid=*value*

timerid=*value*
Timer identifier.

1AB : HKWD SYSC GETTIMER

This event is recorded by the **gettimer** subroutine.

Recorded Data

gettimer timer_type=*value*

timer_type=*value*
Timer type.

1AC : HKWD SYSC INCINTERVAL

This event is recorded by the **incinterval** subroutine.

Recorded Data

incinterval timerid=*value*

timerid=*value*
Timer identifier.

1AD : HKWD SYSC RESTIMER

This event is recorded by the **restimer** subroutine.

Recorded Data

restimer timer_type=*value*

timer_type=*value*
Timer type.

1AE : HKWD SYSC RESABS

This event is recorded by the **resabs** subroutine.

Recorded Data

resabs timer_type=*value*

timer_type=*value*
Timer type.

1AF : HKWD SYSC RESINC

This event is recorded by the **resinc** subroutine.

Recorded Data

resinc timer_type=*value*

timer_type=*value*
Timer type.

1B0 : HKWD VMM ASSIGN

This event is recorded by the virtual memory manager.

Recorded Data

VMM page assign: V.S=*value.value* **ppage=***value* *segment state*	Assign a real page frame to a segment
V.S=*value.value*	Virtual page number and virtual memory identifier
ppage=*value*	Real page frame number
segment state	Segment state information:
WS	Working storage
WS_delete	Working storage with delete pending
delete_pending	
delete_in_progress	
delete_when_iodone	
working_storage	
client_segment	
persistent_storage	
journalled	
log	
deferred_update	
system_segment	
pta_segment	
hidden	
commit_in_progress	
modified	
(type 0)	Page-protection bits = 00
(type 1)	Page-protection bits = 01
(type 2)	Page-protection bits = 02

(type 3)
Page-protection bits = 03.

1B1 : HKWD VMM DELETE

This event is recorded by the virtual memory manager.

Recorded Data

VMM page delete: V.S=*value.value* **ppage=***value segment state*
Delete real page frame from a segment

V.S=*value.value*
Virtual page number and virtual memory identifier

ppage=*value*
Real page frame number

segment state
Segment state information.

1B2 : HKWD VMM PGEXCT

This event is recorded by the virtual memory manager.

Recorded Data

VMM pagefault: V.S=*value.value segment state*
Page fault (other than protection fault or hardware lock-miss faults)

V.S=*value.value*
Virtual page number and virtual memory identifier

segment state
Segment state information.

1B3 : HKWD VMM PROTEXCT

This event is recorded by the virtual memory manager.

Recorded Data

VMM protection fault: V.S=*value.value* **ppage=***value segment state*
Page-protection fault

V.S=*value.value*
Virtual page number and virtual memory identifier

ppage=*value*
Real page frame number

segment state
Segment state information.

1B4 : HKWD VMM LOCKEXCT

This event is recorded by the virtual memory manager.

Recorded Data

VMM lockmiss: V.S=*value.value* **ppage=***value segment state*
Hardware lock miss

V.S=*value.value*
Virtual page number and virtual memory identifier

ppage=*value*
Real page frame number

segment state
Segment state information.

1B5 : HKWD VMM RECLAIM

This event is recorded by the virtual memory manager.

Recorded Data

VMM reclaim: V.S=*value.value* **ppage=***value segment state*
Reclaim a page in the I/O state

V.S=*value.value*
Virtual page number and virtual memory identifier

ppage=*value*
Real page frame number

segment state
Segment state information.

1B6 : HKWD VMM GETPARENT

This event is recorded by the virtual memory manager.

Recorded Data

VMM getparent: V.S=*value.value* **ppage=***value segment state*
Move a page from the parent segment to the child segment

V.S=*value.value*
Virtual page number and virtual memory identifier

ppage=*value*
Real page frame number

segment state
Segment state information.

1B7 : HKWD VMN COPYPARENT

This event is recorded by the virtual memory manager.

Recorded Data

VMM copyparent: V.S=*value.value* **ppage=***value segment state*

V.S=*value.value*
Virtual page number and virtual memory identifier

ppage=*value*
Real page frame number

segment state
Segment state information.

1B8 : HKWD VMN VMAP

This event is recorded by the virtual memory manager.

Recorded Data

VMM vmapped page: V.S=*value.value* **ppage=***value segment state*
Page fault on a page mapped from the source segment to a target segment

V.S=*value.value*
Virtual page number and virtual memory identifier

ppage=*value*
Real page frame number

segment state
Segment state information.

1B9 : HKWD VMN ZFOD

This event is recorded by the virtual memory manager.

Recorded Data

VMM zero filled page: V.S=*value.value* **ppage=***value segment state*
Zero-filled on the demand page fault

V.S=*value.value*
Virtual page number and virtual memory identifier

ppage=*value*
Real page frame number

segment state
Segment state information.

1BA : HKWD VMN SIO

This event is recorded by the virtual memory manager.

Recorded Data

VMM start io: V.S=*value.value* **ppage=***value segment state* **bp=***value bflags*
Start I/O for a page

V.S=*value.value*
Virtual page number and virtual memory identifier

ppage=*value*
Real page frame number

segment state
Segment state information

bp=*value*
Buffer pointer

bflags
Buffer flags:

B_READ
Pagein operation

B_WRITE
Pageout operation.

1BB : HKWD VMM SEGCREATE

This event is recorded by the virtual memory manager.

Recorded Data

VMM segment creation: S=*value segment state*
Creation of a virtual memory object

S=*value*
Virtual memory object identifier

segment state
Segment state information.

1BC : HKWD VMM SEGDELETE

This event is recorded by the virtual memory manager.

Recorded Data

VMM segment deletion: S=*value segment state*
Deletion by the virtual memory manager

S=*value*
Virtual memory object identifier

segment state
Segment state information.

1BD : HKWD VMM DALLOC

This event is recorded by the virtual memory manager.

Recorded Data

NOWRAP>**VMM disk allocation: V.S=***value.value* **dblk=***dblk segment state* **pdtx/devid=***value*	Logical disk block allocation
V.S=*value.value*	Virtual page number and virtual memory object identifier
dblk=*dblk*	Logical disk block number
segment state	Segment state information

pdtx/devid=*value*
: Paging device table index (file system) or device ID (paging space).

1BE : HKWD VMM PFEND

This event is recorded by the virtual memory manager.

Recorded Data

VMM page fault end: V.S=*V.S* **ppage=***value segment state* **error=***error* **bflag=***bflag*	Virtual memory manager I/O done
V.S=*V.S*	Virtual page number and virtual memory identifier
ppage=*value*	Real page frame number
segment state	Segment state information
error=*error*	Exception value
bflag=*bflag*	Possible buffer flags:
B_READ	Pagein operation

B_WRITE
: Pageout operation.

1BF : HKWD VMM EXCEPT

This event is recorded by the virtual memory manager.

Recorded Data

VMM exception: sregval=*sregval* **vaddr=***vaddr segment state* **error=***error* **pid=***pid*	Exception within the virtual memory manager
sregval=*sregval*	Segment register value
vaddr=*vaddr*	Virtual address
segment state	Segment state information
error=*error*	Exception value

pid=*pid*
: Process ID of the process receiving the exception.

Trace Hook IDs: 1C8 through 1CE

The following trace hook IDs are stored in the **/usr/include/sys/trchkid.h** file.

1C8 : HKWD DD PPDD

The event is recorded by the parallel printer device driver.

Recorded Data

Event:

PPDD entry_open: errno: *errno* **devno:** *devno* **rwflag:** *rwflag* **chan:** *chan* **ext:** *ext* **flags:** *open flags*

PPDD exit_open: errno: *errno* **devno:** *devno*

PPDD entry_close: errno: *errno* **devno:** *devno*

PPDD exit_close: errno: *errno* **devno:** *devno*

PPDD entry_read: errno: *errno* **devno:** *devno*

PPDD exit_read: errno: *errno* **devno:** *devno*

PPDD entry_write: errno: *errno* **devno:** *devno* **resid:** *resid* **iovcnt:** *iovcnt* **offset:** *offset* **fmode:** *fmode*

PPDD exit_write: errno: *errno* **devno:** *devno*

PPDD entry_ioctl: errno: *errno* **devno:** *devno* **op:** *ioctl op* **flag:** *dev flag* **chan:** *0* **ext:** *0*

PPDD exit_ioctl: errno: *errno* **devno:** *devno*

errno: *errno*	Error number
devno: *devno*	Major and minor device number
rwflag: *rwflag*	Passed into the device driver to indicate how the device is being used
chan: *chan*	Channel
ext: *ext*	Extension
op: *ioctl op*	Command used in ioctl
flag: *dev flag*	Current status of the device driver
flags: *open flags*	Device flags at open
resid: *resid*	Count left to be sent out
offset: *offset*	Offset into data buffer
iovcnt: *iovcnt*	Number of output buffers
	fmode: *fmode* Type of open.

1C9 : HKWD DD CDDD

This event is recorded by the cd-rom device driver.

Recorded Data

Event:

CDDD entry_open: errno: *errno* **devno:** *devno* **rwflag:** *rwflag* **chan:** *chan* **ext:** *ext*

CDDD exit_open: errno: *errno* **devno:** *devno*

CDDD entry_close: errno: *errno* **devno:** *devno*

CDDD exit_close: errno: *errno* **devno:** *devno*

CDDD entry_read: errno: *errno* **devno:** *devno*

CDDD exit_read: errno: *errno* **devno:** *devno*

CDDD entry_ioctl: errno: *errno* **devno:** *devno* **op:** *ioctl op* **flag:** *ioctl flag* **chan:** *chan* **ext:** *ext*

CDDD exit_ioctl: errno: *errno* **devno:** *devno*

CDDD entry_config: errno: *errno* **devno:** *devno* **op:** *config op*

CDDD exit_config: errno: *errno* **devno:** *devno*

CDDD entry_strategy: errno: *errno* **devno:** *devno* **bp:** *bp* **flags:** *strategy flags* **block:** *block* **bcount:** *bcount*

CDDD exit_strategy: errno: *errno* **devno:** *devno*

CDDD entry_bstart: errno: *errno* **devno:** *devno* **bp:** *bp* **pblock:** *pblock* **bcount:** *bcount bflags*

CDDD exit_bstart: errno: *errno* **devno:** *devno*

CDDD entry_iodone: errno: *errno* **devno:** *devno*

CDDD exit_iodone: errno: *errno* **devno:** *devno*

CDDD iodone: *device name* **bp:** *bp*

errno: *errno*	Error number
devno: *devno*	Major and minor device number
rwflag: *rwflag*	Mode of open
chan: *chan*	Channel
ext: *ext*	Extension
op: *ioctl op*	ioctl operation to perform
flag: *ioctl flag*	Memory address
op: *config op*	Configuration operation to perform
bp: *bp*	Buffer pointer
flags: *strategy flags*	Buffer flags from **buf** structure
block: *block*	Block number on device
bcount: *bcount*	Number of bytes to transfer
pblock: *pblock*	Block number on device

bflags Buffer flags are defined in the **sys/buf.h** file.

1CA : HKWD DD TAPEDD

This event is recorded by the tape device driver.

Recorded Data

Event:

TAPEDD entry_open: errno: *errno* **devno:** *devno* **rwflag:** *rwflag* **chan:** *chan* **ext:** *ext*

TAPEDD exit_open: errno: *errno* **devno:** *devno*

TAPEDD entry_close: errno: *errno* **devno:** *devno*

TAPEDD exit_close: errno: *errno* **devno:** *devno*

TAPEDD entry_read: errno: *errno* **devno:** *devno*

TAPEDD exit_read: errno: *errno* **devno:** *devno*

TAPEDD entry_write: errno: *errno* **devno:** *devno*

TAPEDD exit_write: errno: *errno* **devno:** *devno*

TAPEDD entry_ioctl: errno: *errno* **devno:** *devno* **op:** *ioctl op* **flag:** *ioctl flag* **chan:** *chan* **ext:** *ext*

TAPEDD exit_ioctl: errno: *errno* **devno:** *devno*

TAPEDD entry_config: errno: *errno* **devno:** *devno* **op:** *config op*

TAPEDD exit_config: errno: *errno* **devno:** *devno*

TAPEDD entry_cstart: errno: *0* **devno:** *devno* **command:** *cstart cmd* **baddress:** *baddress* **bcount:** *bcount*

TAPEDD exit_cstart: errno: *errno* **devno:** *devno*

TAPEDD entry_iodone: errno: *0* **devno:** *devno* **command:** *iodone cmd* **baddress:** *baddress* **bcount:** *bcount*

TAPEDD exit_iodone: errno: *errno* **devno:** *devno*

TAPEDD iodone: *device name* **bp:** *bp*

errno: *errno*	Error number
devno: *devno*	Major and minor device number
rwflag: *rwflag*	Possible values:
FREAD	Device opened read-only
FWRITE	Device opened read-write
chan: *chan*	Channel
ext: *ext*	Extension
op: *ioctl op*	ioctl operation
flag: *ioctl flag*	Address of users argument structure
op: *config op*	Possible values:
CFG_INIT	Configures the device
CFT_TERM	Unconfigures the device
bcount: *bcount*	Number of bytes to transfer
command: *cstart cmd*	Low-order byte contains SCSI command issued to the drive
baddress: *baddress*	Buffer address where information is transferred to and from the device; zero for commands that do not transfer data
	command: *iodone cmd* Low-order byte contains SCSI command issued to the drive
	bp: *bp* Buffer pointer.

1CD : HKWD DD ENTDD

This event is recorded by the ethernet device handler to track the various phases of data transfer within the device handler.

Recorded Data

Event:

Ethernet: enque kernel data *device name* **mbuf=***mbuf* **count=***count* **channel=***channel*

Ethernet: enque user data *device name* **mbuf=***mbuf* **count=***count* **channel=***channel*

Ethernet: receive overflow *device name* **mbuf=***mbuf* **count=***count* **channel=***channel*

Ethernet: transmit done *device name* **mbuf=***mbuf* **count=***count* **channel=***channel*

Ethernet: return form read *device name* **mbuf=***mbuf* **count=***count* **channel=***channel*

Ethernet: write *device name* **mbuf=***mbuf* **count=***count* **channel=***channel*

Ethernet: transmit interrupt *device name* **mbuf=***mbuf* **count=***count* **channel=***channel*

Ethernet: receive interrupt *device name* **mbuf=***mbuf* **count=***count* **channel=***channel*

device name
: The **/dev** entry point for this device

mbuf=*mbuf*
: Address of the mbuf that contains the user data

count=*count*
: Number of bytes of user data to be transferred

channel=*channel*
: Channel number of the process that opened the device.

1CE : HKWD DD TOKDD

This event is recorded by the token ring device driver.

Recorded Data

Event:

Token Ring: enque kernel data *device name* **mbuf=***mbuf* **count=***count* **channel=***channel*

Token Ring: enque user data *device name* **mbuf=***mbuf* **count=***count* **channel=***channel*

Token Ring: receive overflow *device name* **mbuf=***mbuf* **count=***count* **channel=***channel*

Token Ring: transmit done *device name* **mbuf=***mbuf* **count=***count* **channel=***channel*

Token Ring: return form read *device name* **mbuf=***mbuf* **count=***count* **channel=***channel*

Token Ring: write *device name* **mbuf=***mbuf* **count=***count* **channel=***channel*

Token Ring: transmit interrupt *device name* **mbuf=***mbuf* **count=***count* **channel=***channel*

Token Ring: receive interrupt *device name* **mbuf=***mbuf* **count=***count* **channel=***channel*

device name	The **/dev** entry point for this device
mbuf=*mbuf*	Address of the mbuf which contains the user data
count=*count*	Number of bytes of user data to be transferred

channel=*channel*
Channel number of the process that opened the device.

Trace Hook IDs: 1CF through 211

The following trace hook IDs are stored in the **/usr/include/sys/trchkid.h** file.

1CF : HKWD DD C327DD

This event is recorded by the 3270 Connection Adapter device driver.

Recorded Data

Event:

C327DD entry_open: errno: *errno* **devno:** *devno* **rwflag:** *rwflag* **chan:** *chan* **ext:** *ext*

C327DD exit_open: errno: *errno* **devno:** *devno*

C327DD entry_close: errno: *errno* **devno:** *devno* **chan:** *chan*

C327DD exit_close: errno: *errno* **devno:** *devno*

C327DD entry_read: errno: *errno* **devno:** *devno*

C327DD exit_read: errno: *errno* **devno:** *devno*

C327DD entry_write: errno: *errno* **devno:** *devno* **uiop:** *uiop* **chan:** *chan* **ext:** *ext*

C327DD exit_write: errno: *errno* **devno:** *devno*

C327DD entry_ioctl: errno: *errno* **devno:** *devno* **op:** *ioctl op* **flag:** *ioctl flag* **chan:** *chan* **ext:** *ext*

C327DD exit_ioctl: errno: *errno* **devno:** *devno*

C327DD entry_select: errno: *errno* **devno:** *devno* **event:** *event* **chan:** *chan*

C327DD exit_select: errno: *errno* **devno:** *devno*

C327DD entry_config: errno: *errno* **devno:** *devno* **op:** *config op*

C327DD exit_config: errno: *errno* **devno:** *devno*

C327DD entry_mpx: errno: *errno* **devno:** *devno* **name:** *name* **chan:** *chan*

C327DD exit_mpx: errno: *errno* **devno:** *devno* **name:** *name* **chan:** *chan* **oflag:** *mpx flag*

errno: *errno*	Error number
devno: *devno*	Major and minor device number

rwflag: *rwflag*	Open flags
chan: *chan*	Channel
ext: *ext*	Extension
uiop: *uiop*	**uio** structure pointer
event: *event*	Event specified in the **select** or **poll** subroutine
op: *ioctl op*	Command code specified in the **ioctl** subroutine
flag: *ioctl flag*	Argument code specified in the **ioctl** subroutine
op: *config op*	Command code specified in the **config** subroutine
name: *name*	Path-name extension of the multiplex channel to be allocated

oflag: *mpx flag*
Unused.

1D1 : HKWD RAS ERRLG

This event is recorded by the **/dev/error** file.

Recorded Data

Event:

ERRLG erropen: *errno*

ERRLG errclose: *errno*

ERRLG errioctl: *errno device name* **ERRIOC_STOP**

ERRLG errioctl: *errno device name* **ERRIOC_SYNC**

ERRLG errread: bad erec_length *length* **bytes**

ERRLG errread: *errno*

ERRLG errwrite: *errno*

ERRLG errput

ERRLG errput: buffer overflow: state=*state*

ERRLG errdd: lockl from *value* **already locked by** *process*

ERRLG errdd: unlockl from *value* **not locked**

ERRLG errdemon: cannot write to errlog. error id=*error id*

errno	Error number
device name	Device name
length	Length
state=*state*	Possible values:

RDOPEN

SLEEP

STOP

SYNC

process Process name and ID
lockl from *value* Routine that called the **lockl** subroutine
unlockl from *value* Routine that called the **unlockl** subroutine

error id=*error id*
Error identifier.

1D2 : HKWD RAS DUMP

This event is recorded by the dump device driver.

Recorded Data

Event:

DUMP dmpopen : *errno device name*

DUMP dmpioctl : *errno device name* **DMPSET_PRIM**

DUMP dmpioctl : *errno device name* **DMPSET_SEC**

DUMP dmpioctl : *errno device name* **DMPNOW_PRIM**

DUMP dmpioctl : *errno device name* **DMPNOW_SEC**

DUMP dmpdump : DUMPINIT *device name*

DUMP dmpdump : DUMPSTART *device name*

DUMP dmpdump : DUMPWRITE *device name*

DUMP dmpdump : DUMPEND *device name*

DUMP dmpdump : DUMPTERM *device name*

DUMP dmpdump : DUMPQUERY *device name*

DUMP dmpadd : calling func is *function*

DUMP dmp : return: *errno*

DUMP dmpdel : calling func is *function*

DUMP dmpdel : return: *errno*

DUMP dmp_do : PRIMARY

DUMP dmp_do : SECONDARY

DUMP dmp_do : return: *errno*

DUMP dmpwrcdt : ptr=*wrcdt ptr* **length=***wrcdtlength*

DUMP dump_op : return: *errno*

DUMP dmpnull : DUMPINIT

DUMP dmpnull : DUMPSTART

DUMP dmpnull : DUMPWRITE

DUMP dmpnull : DUMPEND

DUMP dmpnull : DUMPTERM

DUMP dmpnull : DUMPQUERY

DUMP dmpfile : DUMPINIT

DUMP dmpfile : DUMPSTART

DUMP dmpfile : DUMPWRITE

DUMP dmpfile : DUMPEND

DUMP dmpfile : DUMPTERM

DUMP dmpfile : DUMPQUERY

errno	Error number
device name	Name of dump device
function	Name of function calling the **dmp_add** subroutine or **dmp_del** subroutine
ptr=*wrcdt ptr*	Pointer to Component Dump Table to be written
	length=*wrcdt length* Length of Component Dump Table to be written.

1F0 : HKWD SYSC SETTIMER

This event is recorded by the **settimer** subroutine.

Recorded Data

settimer timer_type *timer type*

timer_type *timer type*

Type of timer.

200 : HKWD KERN RESUME

This event is recorded by the **resume** subroutine.

Recorded Data

resume *process name*
resume interrupt process mst=*mst*

process name	Process name of the resumed thread.
mst	MST of the resumed thread.

20E: HKWD KERN LOCKL

This event is recorded by the **lockl** kernel service.

Recorded Data

lockl lock address=*lock address* **lock value=***lock value*
return address=*return address* **flags=***flags*

lock address Address of the lock word
lock value Content of the lock word
return address Return address of the caller
flags Flags parameter.

20F: HKWD KERN UNLOCKL

This event is recorded by the **unlockl** kernel service.

Recorded Data

unlockl lock address=*lock address* **lock value=***lock value* **return address=***return address*

lock address Address of the lock word
lock value Content of the lock word
return address
Return address of the caller.

211 : HKWD NFS VOPSRW

This event is recorded to the read/write vnop op for NFS client.

Recorded Data

Event:

NFS_READ *filename* **count=***count* **offset=***offset* **sid=***sid* Client NFS read call entry
NFS_WRITE *filename* **count=***count* **offset=***offset* **sid=***sid* Client NFS write call entry

filename
File path name

count=*count*

offset=*offset*

sid=*sid*

Trace Hook IDs: 212 through 220

The following trace hook IDs are stored in the **/usr/include/sys/trchkid.h** file.

212 : HKWD NFS VOPS

This event is recorded by the client NFS routine entry points.

Recorded Data

Event:

NFS_LOOKUP *filename*

NFS_CREATE *filename*

NFS_REMOVE *filename*

NFS_LINK *filename*

NFS_RENAME from: *filename*

NFS_RENAME to: *filename*

NFS_MKDIR *filename*

NFS_RMDIR *filename*

NFS_SYMLINK from: *filename*

NFS_SYMLINK to: *filename*

NFS_SELECT *filename*

NFS_LOOKUP vnode=*vnode*

NFS_OPEN *filename*

NFS_CLOSE *filename*

NFS_IOCTL *filename*

NFS_GETATTR *filename*

NFS_SETATTR *filename*

NFS_ACCESS *filename*

NFS_CREATE *filename*

NFS_REMOVE *filename*

NFS_LINK *filename*

NFS_RENAME *filename*

NFS_MKDIR *filename*

NFS_RMDIR *filename*

NFS_READDIR *filename*

NFS_SYMLINK *filename*

NFS_READLINK *filename*

NFS_FSYNC *filename*

NFS_INACTIVE *filename*

NFS_BMAP *filename*

NFS_BADOP

NFS_STRATEGY *filename*

NFS_LOCKCTL *filename*

NFS_NOOP

NFS_CMP *filename*

filename File path name

vnode=*vnode*
v_node.

213 : HKWD NFS RFSRW

This event is recorded by the server NFS read/write routines.

Recorded Data

RFS_READ seqno=*seqno filename vnode* **count=***count* **offset=***offset*	Server read request
RFS_WRITE seqno=*seqno filename vnode* **count=***count* **offset=***offset*	Server write request
seqno=*seqno*	Sequence number to match client call
filename	File path name
vnode	v_node of file
count=*count*	Number of bytes to read or write
	offset=*offset* Offset in file to read or write.

214 : HKWD NFS RFS

This event is recorded by the server NFS routine entry points.

Recorded Data

Event:

RFS_LOOKUP *filename*

RFS_LOOKUP *filename*

RFS_CREATE *filename*

RFS_REMOVE *filename*

RFS_RENAME from: *filename*

RFS_RENAME to: *filename*

RFS_LINK *filename*

RFS_SYMLINK from: *filename*

RFS_SYMLINK to: *filename*

RFS_MKDIR *filename*

RFS_RMDIR *filename*

RFS_NULL seqno=*seqno*

RFS_GETATTR seqno=*seqno filename*

RFS_SETATTR seqno=*seqno filename*

RFS_ERROR

RFS_LOOKUP seqno=*seqno filename*

RFS_READLINK seqno=*seqno filename*

RFS_CREATE seqno=*seqno filename*

RFS_REMOVE seqno=*seqno filename*

RFS_RENAME seqno=*seqno filename filename*

RFS_LINK seqno=*seqno filename filename*

RFS_SYMLINK seqno=*seqno filename*

RFS_MKDIR seqno=*seqno filename*

RFS_RMDIR seqno=*seqno filename*

RFS_READDIR seqno=*seqno filename*

RFS_STATFS seqno=*seqno filename*

filename File path name

seqno=*seqno*
Sequence number to match client call.

215 : HKWD NFS DISPATCH

This event is recorded by the server dispatch routine entry and exit.

Recorded Data

Event:

RFS_DISP_ENTRY seqno=*seqno* **client=***client*
NOWRAP>**RFS_DISP_EXIT seqno=***seqno* **client=***client*
dispcode

seqno=*seqno*	Sequence number to match calls to client-side request
client=*client*	IP address of client
dispcode	Routine called on the server:

NULL

GETATTR

SETATTR

LOOKUP

READLINK

READ

WRITE

CREATE

REMOVE

RENAME

LINK

SYMLINK

MKDIR

RMDIR

READDIR

STATFS

216 : HKWD NFS CALL

This event is recorded by the NFS call routine entry and exit.

Recorded Data

Event:

NFS_CALL_ENTRY seqno=*seqno* **server=***server*

NFS_CALL_EXIT seqno=*seqno* **server=***server*

seqno=*seqno* Sequence number to track call on server

server=*server*
Server IP address.

218 : HKWD RPC LOCKD

This event is recorded by the RPC **lockd** routine entry points.

Recorded Data

Event:

LOCKD_KLM_PROG proc=*proc* **pid=***pid* **cookie=***cookie* **port=***port*	Entry point for remote lock requests coming from the kernel
LOCKD_NLM_REQUEST proc=*proc* **to** *addr* **cookie=***cookie* **pid=***pid*	Entry point for incoming lock request on the network
LOCKD_NLM_RESULTS proc=*proc* **to** *addr* **cookie=***cookie* **result=***result*	Entry point for responses coming over the network
LOCKD_KLM_REPLY proc=*proc* **stat=***stat* **cookie=***cookie*	Entry point for lockd reply to kernel
LOCKD_NLM_REPLY proc=*proc* **to** *addr* **stat=***stat* **cookie=***cookie*	Entry point for lockd reply to network
LOCKD_NLM_CALL proc=*proc* **cookie=***cookie* **pid=***pid* **retransmit=***retransmit*	Entry point for sending lock request over the network
LOCKD_CALL_UDP to *addr* **proc=***proc* **program=***program* **version=***version*	Entry point for send udp request for RPC.lockd.
proc=*proc*	RPC procedure number
pid=*pid*	Process ID
cookie=*cookie*	Internal RPC.lockd counter
port=*port*	Socket port
to *addr*	Internet address
result=*result*	Result for a previous request
stat=*stat*	RPC.lockd reply status
retransmit=*retransmit*	Value of retransmit flag
program=*program*	RPC program number
	version=*version* RPC version number.

220 : HKWD DD FDDD

This event is recorded by the diskette device driver.

Recorded Data

Event:

FDDD entry_open: errno: *errno* **devno:** *devno* **rwflag:** *rwflag* **chan:** *chan* **ext:** *ext*

FDDD exit_open: errno: *errno* **devno:** *devno*

FDDD entry_close: errno: *errno* **devno:** *devno*

FDDD exit_close: errno: *errno* **devno:** *devno*

FDDD entry_read: errno: *errno* **devno:** *devno*

FDDD exit_read: errno: *errno* **devno:** *devno*

FDDD entry_write: errno: *errno* **devno:** *devno*

FDDD exit_write: errno: *errno* **devno:** *devno*

FDDD entry_ioctl: errno: *errno* **devno:** *devno* **op:** *ioctl op* **flag:** *ioctl flag* **chan:** *chan* **ext:** *ext*

FDDD exit_ioctl: errno: *errno* **devno:** *devno*

FDDD entry_select: errno: *errno* **devno:** *devno*

FDDD exit_select: errno: *errno* **devno:** *devno*

FDDD entry_config: errno: *errno* **devno:** *devno* **op:** *config op*

FDDD exit_config: errno: *errno* **devno:** *devno*

FDDD entry_strategy: errno: *errno* **devno:** *devno* **bp:** *bp* **flags:** *strategy flags* **block:** *block* **bcount:** *bcount*

FDDD exit_strategy: errno: *errno* **devno:** *devno*

FDDD entry_mpx: errno: *errno* **devno:** *devno*

FDDD exit_mpx: errno: *errno* **devno:** *devno* **name:** *name* **chan:** *chan* **oflag:** *mpx oflag*

FDDD entry_revoke: errno: *errno* **devno:** *devno*

FDDD exit_revoke: errno: *errno* **devno:** *devno*

FDDD entry_intr: errno: *errno* **devno:** *devno*

FDDD exit_intr: errno: *errno* **devno:** *devno*

FDDD entry_bstart: errno: *errno* **devno:** *devno* **bp:** *bp* **pblock:** *pblock* **bcount:** *bcount bflags*

FDDD exit_bstart: errno: *errno* **devno:** *devno*

FDDD entry_cstart: errno: *errno* **devno:** *devno*

FDDD exit_cstart: errno: *errno* **devno:** *devno*

FDDD entry_iodone: errno: *errno* **devno:** *devno*

FDDD exit_iodone: errno: *errno* **devno:** *devno*

FDDD iodone: *device name* **bp:** *bp*

errno: *errno*	Error number
devno: *devno*	Major and minor device number
rwflag: *rwflag*	Possible values:

FREAD	Device is opened read-only
FWRITE	Device is opened read-write.
chan: *chan*	Channel
ext: *ext*	Extension
op: *ioctl op*	ioctl operation
flag: *ioctl flag*	Address of users argument structure
op: *config op*	Possible values:
CFG_INIT	Configures the device
CFG_TERM	Unconfigures the device.
bp: *bp*	Buffer pointer
flags: *strategy flags*	Buffer flags field in the **buf** structure
block: *block*	Physical block number
bcount: *bcount*	Number of bytes to transfer
name: *name*	Path-name extension of multiplex channel to be allocated
oflag: *mpx flag*	
pblock: *pblock*	Physical block
	bflags Buffer flags are defined in the **sys/buf.h** file.

Trace Hook IDs: 221 through 223

The following trace hook IDs are stored in the **/usr/include/sys/trchkid.h** file.

221 : HKWD DD SCDISKDD

This event is recorded by the SCSI device driver

Recorded Data

SCDISKDD entry_open: errno: *errno* **devno:** *devno* **rwflag:** *rwflag* **chan:** *chan* **ext:** *ext*

SCDISKDD exit_open: errno: *errno* **devno:** *devno*

SCDISKDD entry_close: errno: *errno* **devno:** *devno*

SCDISKDD exit_close: errno: *errno* **devno:** *devno*

SCDISKDD entry_read: errno: *errno* **devno:** *devno*

SCDISKDD exit_read: errno: *errno* **devno:** *devno*

SCDISKDD entry_write: errno: *errno* **devno:** *devno*

SCDISKDD exit_write: errno: *errno* **devno:** *devno*

SCDISKDD entry_ioctl: errno: *errno* **devno:** *devno* **op:** *ioctl op* **flag:** *ioctl flag* **chan:** *chan* **ext:** *ext*

SCDISKDD exit_ioctl: errno: *errno* **devno:** *devno*

SCDISKDD entry_config: errno: *errno* **devno:** *devno* **op:** *config op*

SCDISKDD exit_config: errno: *errno* **devno:** *devno*

SCDISKDD entry_strategy: errno: *errno* **devno:** *devno* **bp:** *bp* **flags:** *strategy flags* **block:** *block* **bcount:** *bcount*

SCDISKDD exit_strategy: errno: *errno* **devno:** *devno*

SCDISKDD entry_bstart: errno: *errno* **devno:** *devno* **bp:** *bp* **pblock:** *pblock* **bcount:** *bcount bflags*

SCDISKDD exit_bstart: errno: *errno* **devno:** *devno*

SCDISKDD entry_iodone: errno: *errno* **devno:** *devno*

SCDISKDD exit_iodone: errno: *errno* **devno:** *devno* **sc_bufp:** *sc bufp*

SCDISKDD coalesce: (*bp,sc bp*)

SCDISKDD iodone: errno: *errno* **devno:** *devno* **bp:** *bp*

errno: *errno*	Error number
devno: *devno*	Major and minor device number
rwflag: *rwflag*	Possible values:
FREAD	Device is opened read-only
FWRITE	Device is opened read-write.
chan: *chan*	Channel:
For open: always zero	
For ioctl: DKERNEL if called by kernel process	
ext: *ext*	Extension:
SC_DIAGNOSTIC	Open in diagnostic mode
SC_RETAIN_RESERVATION	Do not release reservation on close
SC_FORCED_OPEN	Reset device before opening.
op: *ioctl op*	Possible values:
IOCINFO	Get information about the device
DKIORDSE	Issue **read** command and return sense data if error occurs
DKIOWRSE	Issue **write** command and return sense data if error occurs
DKIOCMD	Issue pass-through command (user-defined) to the device.
flag: *ioctl flag*	Address of the user's argument structure
op: *config op*	Possible values:
CFG_INIT	Configure the device
CFG_TERM	Unconfigure the device.
bp: *bp*	Buffer pointer

flags: *strategy flags*

block: *block*

bcount: *bcount*	Number of bytes to be read or written
pblock: *pblock*	Physical block
bflags	Buffer flags are defined in the **sys/buf.h** file

sc_bufp: *sc bufp* SCSI buffer pointer

(*bp, sc bp***)**
Parameters used to issue this command to the SCSI adapter driver:

bp Buffer pointer

sc bp Associated SCSI buffer pointer.

222 : HKWD DD BADISKDD

This event is recorded by the bus-attached hard disk device driver.

Recorded Data

BADDD entry_open: errno: *errno* **devno:** *devno* **rwflag:** *rwflag* **chan:** *chan* **ext:** *ext*

BADDD exit_open: errno: *errno* **devno:** *devno*

BADDD entry_close: errno: *errno* **devno:** *devno*

BADDD exit_close: errno: *errno* **devno:** *devno*

BADDD entry_read: errno: *errno* **devno:** *devno*

BADDD exit_read: errno: *errno* **devno:** *devno*

BADDD entry_write: errno: *errno* **devno:** *devno*

BADDD exit_write: errno: *errno* **devno:** *devno*

BADDD entry_ioctl: errno: *errno* **devno:** *devno* **op:** *ioctl op* **flag:** *ioctl flag* **chan:** *chan* **ext:** *ext*

BADDD exit_ioctl: errno: *errno* **devno:** *devno*

BADDD entry_config: errno: *errno* **devno:** *devno* **op:** *config op*

BADDD exit_config: errno: *errno* **devno:** *devno*

BADDD entry_strategy: errno: *errno* **devno:** *devno* **bp:** *bp* **flags:** *strategy flags* **block:** *block* **bcount:** *bcount*

BADDD exit_strategy: errno: *errno* **devno:** *devno*

BADDD entry_intr: errno: *errno* **devno:** *devno*

BADDD exit_intr: errno: *errno* **devno:** *devno*

BADDD entry_bstart: errno: *errno* **devno:** *devno* **bp:** *bp* **pblock:** *pblock* **bcount:** *bcount bflags*

BADDD exit_bstart: errno: *errno* **devno:** *devno*

errno: *errno* Error number
devno: *devno* Major and minor device number

rwflag: *rwflag*	Possible values:
FREAD	Device is opened read-only
FWRITE	Device is opened read-write.
chan: *chan*	Channel
ext: *ext*	Extension
op: *ioctl op*	
flag: *ioctl flag*	Address of the users argument structure
op: *config op*	Possible values:
CFG_INIT	Configure the device
CFG_TERM	Unconfigure the device.
bp: *bp*	Buffer pointer
flags: *strategy flags*	Buffer flags field in the **buf** structure
block: *block*	Physical block
bcount: *bcount*	Number of bytes to read or write
pblock: *pblock*	Physical block

bflags Buffer flags are defined in the **sys/buf.h** file.

223 : HKWD DD SCSIDD

This event is recorded by the SCSI adapter driver.

Recorded Data

SCSIDD entry_open: errno: *errno* **devno:** *devno* **rwflag:** *rwflag* **chan:** *chan* **ext:** *ext*

SCSIDD exit_open: errno: *errno* **devno:** *devno*

SCSIDD entry_close: errno: *errno* **devno:** *devno*

SCSIDD exit_close: errno: *errno* **devno:** *devno*

SCSIDD entry_read: errno: *errno* **devno:** *devno*

SCSIDD exit_read: errno: *errno* **devno:** *devno*

SCSIDD entry_write: errno: *errno* **devno:** *devno*

SCSIDD exit_write: errno: *errno* **devno:** *devno*

SCSIDD entry_ioctl: errno: *errno* **devno:** *devno* **op:** *ioctl op* **flag:** *ioctl flag* **chan:** *chan* **ext:** *ext*

SCSIDD exit_ioctl: errno: *errno* **devno:** *devno*

SCSIDD entry_select: errno: *errno* **devno:** *devno*

SCSIDD exit_select: errno: *errno* **devno:** *devno*

SCSIDD entry_config: errno: *errno* **devno:** *devno* **op:** *config op*

SCSIDD exit_config: errno: *errno* **devno:** *devno*

SCSIDD strategy: bp: *bp*

SCSIDD exit_strategy: errno: *errno* **devno:** *devno*

SCSIDD entry_mpx: errno: *errno* **devno:** *devno*

SCSIDD exit_mpx: errno: *errno* **devno:** *devno* **name:** *name* **chan:** *chan* **oflag:** *mpx flag*

SCSIDD entry_revoke: errno: *errno* **devno:** *devno*

SCSIDD exit_revoke: errno: *errno* **devno:** *devno*

SCSIDD entry_intr: errno: *errno* **devno:** *devno*

SCSIDD exit_intr: errno: *errno* **devno:** *devno*

SCSIDD entry_bstart: *device name* **bp:** *bp* **pblock:** *pblock* **bcount:** *bcount bflags*

SCSIDD exit_bstart: errno: *errno* **devno:** *devno*

SCSIDD entry_cstart: errno: *errno* **devno:** *devno*

SCSIDD exit_cstart: errno: *errno* **devno:** *devno*

SCSIDD entry_iodone: errno: *errno* **devno:** *devno*

SCSIDD exit_iodone: errno: *errno* **devno:** *devno*

SCSIDD scsi_intr: errno: *errno* **devno:** *devno* **sc_bufp:** *sc bufp*

SCSIDD coalesce: (*bp,sc bp*)

SCSIDD iodone: *device name* **bp:** *bp filename*

errno: *errno*	Error number
devno: *devno*	Major and minor device number
rwflag: *rwflag*	Possible values:
FREAD	Device is opened read-only
FWRITE	Device is opened read-write.
chan: *chan*	Channel
ext: *ext*	Extension
op: *ioctl op*	ioctl operation
flag: *ioctl flag*	Address of the user's argument structure
op: *config op*	Possible values:
CFG_INIT	Configure the device
CFG_TERM	Unconfigure the device.
bp: *bp*	Buffer pointer
flags: *strategy flags*	Buffer flags field in the **buf** structure
block: *block*	Physical block
bcount: *bcount*	Number of bytes to read or write
name: *name*	Path-name extension of multiplex channel to be allocated
oflag: *mpx flag*	
pblock: *pblock*	Physical block
bflags	Buffer flags are defined in the **sys/buf.h** file

sc_bufp: *sc bufp*

(*bp, sc bp***)**

Parameters used to issue this command:

bp Buffer pointer

sc bp Associated SCSI buffer pointer.

filename

File path name.

Trace Hook IDs: 224 through 226

The following trace hook IDs are stored in the **/usr/include/sys/trchkid.h** file.

224 : HKWD DD MPQPDD

This event is recorded by the Multiprotocol Quad Port (MPQP) device driver.

Recorded Data

MPQPDD entry_open: errno: *errno* **devno:** *devno* **devflag:** *devflag* **chan:** *chan* **p_ext:** *p_ext*

MPQPDD exit_open: break=all. errno: *errno* **devno:** *devno* **Suberror:** *suberror*

MPQPDD entry_close: errno: *errno* **devno:** *devno* **chan:** *chan*

MPQPDD exit_close: errno: *errno* **devno:** *devno* **Suberror:** *suberror*

MPQPDD entry_read: errno: *errno* **devno:** *devno* **bufptr:** *bufptr* **chan:** *chan* **ext:** *ext*

MPQPDD exit_read: errno: *errno* **devno:** *devno* **bufptr:** *bufptr* **chan:** *chan* **status:** *status* **Suberror:** *suberror*

MPQPDD entry_write: errno: *errno* **devno:** *devno* **bufptr:** *bufptr* **chan:** *chan* **ext:** *ext*

MPQPDD exit_write: errno: *errno* **devno:** *devno* **bufptr:** *bufptr* **chan:** *chan* **status:** *status* **Suberror:** *suberror*

MPQPDD entry_ioctl: errno: *errno* **devno:** *devno* **op:** *op* **flag:** *flag* **chan:** *chan* **ext:** *ext*

MPQPDD exit_ioctl: errno: *errno* **devno:** *devno* **Suberror:** *suberror*

MPQPDD entry_select: errno: *errno* **devno:** *devno* **events:** *events* **chan:** *chan*

MPQPDD exit_select: errno: *errno* **devno:** *devno* **reventp:** *reventp* **chan:** *chan* **Suberror:** *suberror*

MPQPDD entry_config: errno: *errno* **devno:** *devno* **op:** *op*

MPQPDD exit_config: errno: *errno* **devno:** *devno* **Suberror:** *suberror*

MPQPDD entry_mpx: errno: *errno* **devno:** *devno*

MPQPDD exit_mpx: errno: *errno* **devno:** *devno* **nameptr:** *nameptr* **chan:** *chan* **openflag:** *openflag* **Suberror:** *suberror*

MPQPDD entry_intr: errno: *errno* **devno:** *devno*

MPQPDD exit_intr: errno: *errno* **devno:** *devno* **status:** *status*

MPQPDD entry_cstart: errno: *errno* **devno:** *devno* **parm1:** *parm1* **parm2:** *parm2* **parm3:** *parm3* **parm4:** *parm4*

MPQPDD exit_cstart: errno: *errno* **devno:** *devno* **parm#:** *parm#* **Parmval:** *Parmval* **Suberror:** *suberror*

MPQPDD entry_halt: errno: *errno* **devno:** *devno*

MPQPDD exit_halt: errno: *errno* **devno:** *devno* **status:** *status*

MPQPDD entry_getstat: errno: *errno* **devno:** *devno* **devflag:** *devflag* **chan:** *chan*

MPQPDD exit_getstat: errno: *errno* **devno:** *devno* **code:** *code* **opt[0]:** *opt[0]* **opt[1]:** *opt[1]* **opt[2]:** *opt[2]*

MPQPDD exit_kread: errno: *errno* **devno:** *devno* **openid:** *openid* **status:** *status* **bufptr:** *bufptr*

MPQPDD exit_kstat: errno: *errno* **devno:** *devno* **openid:** *openid* **code:** *code* **opt[0]:** *opt[0]* **opt[1]:** *opt[0]*

MPQPDD exit_ktx_fn: errno: *errno* **devno:** *devno* **openid:** *openid*

MPQPDD entry_chgparm: errno: *errno* **devno:** *devno* **rcv timer:** *rcv timer* **Poll addr:** *Poll addr* **Select addr:** *Select addr*

MPQPDD exit_chgparm: errno: *errno* **devno:** *devno*

MPQPDD entry_start_ar: errno: *errno* **devno:** *devno*

MPQPDD exit_start_ar: errno: *errno* **devno:** *devno*

MPQPDD entry_flushport: errno: *errno* **devno:** *devno*

MPQPDD exit_flushport: errno: *errno* **devno:** *devno*

MPQPDD entry_adaptquery: errno: *errno* **devno:** *devno*

MPQPDD exit_adaptquery: errno: *errno* **devno:** *devno*

MPQPDD entry_query_stat: errno: *errno* **devno:** *devno*

MPQPDD entry_trace_on: errno: *errno* **devno:** *devno*

MPQPDD exit_trace_on: errno: *errno* **devno:** *devno*

MPQPDD entry_stop_port: errno: *errno* **devno:** *devno*

MPQPDD exit_stop_port: errno: *errno* **devno:** *devno*

MPQPDD entry_traceoff: errno: *errno* **devno:** *devno*

MPQPDD exit_traceoff: errno: *errno* **devno:** *devno*

errno: *errno*	Error number
devno: *devno*	Major and minor device number

devflag: *devflag* Device flag
chan: *chan* Channel
p_ext: *p_ext* Pointer to extension
ext: *ext* Extension
bufptr: *bufptr* Buffer pointer
status: *status*
op: *op* ioctl operation
flag: *flag* ioctl **devflag** argument
events: *events* **events** argument for **select**
reventp: *reventp* **reventp** argument for **select**
nameptr: *nameptr* Pointer to channel name

openflag: *openflag*

parm1: *parm1* **parm1** parameter to **cstart**; physical link

parm2: *parm2***parm2** parameter to **cstart**; data flags

parm3: *parm3***parm3** parameter to **cstart**; baud rate

parm4: *parm4*

parm4 parameter to **cstart**; receive data offset

parm#: *parm#* Parameter number
Parmval: *Parmval* Parameter value

opt[0]: *opt[0]*

opt[1]: *opt[1]*

opt[2]: *opt[2]*

openid: *openid*

code: *code*

rcv timer: *rcv timer* Receive timer
Poll addr: *Poll addr* Poll address
Select addr: *Select addr* Select address
Suberror: *suberror* Additional error information:

Adapter number too big.

There is no ACB.

No offlevel intr. structure.

Cannot register interrupt.

No port dds.

Channel too big.

Channel busy.

No mbuf available.

No transmit chain.

Adapter already opened.

Cannot set up POS REG.

Error in uiomove.

Port not open.

Port not started.

Pin code failed.

Add entry failed in devswadd.

Port already opened.

Physical link invalid.

Data protocol invalid.

Baud rate invalid.

None.

225 : HKWD DD X25DD

This event is recorded by the X25 device driver.

Recorded Data

Event:

X25DD entry_open: errno: *errno* **devno:** *devno* **flag:** *flag* **chan:** *chan* **ext:** *ext*

X25DD exit_open: errno: *errno* **devno:** *devno* **Suberror:** *suberror* **chan:** *chan*

X25DD entry_close: errno: *errno* **devno:** *devno* **chan:** *chan*

X25DD exit_close: errno: *errno* **devno:** *devno* **chan:** *chan* **gp_rc:** *gp_rc*

X25DD entry_read: errno: *errno* **devno:** *devno* **chan:** *chan* **ext:** *ext*

X25DD exit_read: errno: *errno* **devno:** *devno* **packet_type:** *packet_type* **session_id:** *sesson_id* **status:** *status*

X25DD entry_write: errno: *errno* **devno:** *devno* **chan:** *chan* **ext:** *ext*

X25DD exit_write: errno: *errno* **devno:** *devno* **packet_type:** *packet_type* **session_id:** *sesson_id* **status:** *status*

X25DD entry_ioctl: errno: *errno* **devno:** *devno* **cmd:** *cmd* **flag:** *flag* **chan:** *chan*

X25DD exit_ioctl: errno: *errno* **devno:** *devno*

X25DD entry_select: errno: *errno* **devno:** *devno* **chan:** *chan* **events:** *events*

X25DD exit_select: errno: *errno* **devno:** *devno* **chan:** *chan* **events:** *events* **reventp:** *reventp*

X25DD entry_config: errno: *errno* **devno:** *devno* **uiop:** *uiop*

X25DD exit_config: errno: *errno* **devno:** *devno*

X25DD entry_mpx: errno: *errno* **devno:** *devno*

X25DD exit_mpx: errno: *errno* **devno:** *devno* **channame:** *channame* **chan:** *chan*

X25DD entry_halt: errno: *errno* **devno:** *devno* **chan:** *chan*

X25DD exit_halt: errno: *errno* **devno:** *devno* **chan:** *chan* **status:** *status* **session_id:** *sesson_id* **session_type:** *session_type*

X25DD entry_get_stat: errno: *errno* **devno:** *devno* **flag:** *flag* **chan:** *chan* **ext:** *ext*

X25DD exit_get_stat: errno: *errno* **devno:** *devno* **block.code:** *block.code* **block.opt 0:** *block.opt 0* **block.opt 1:** *block.opt 1*

X25DD entry_iocinfo: errno: *errno* **devno:** *devno*

X25DD exit_iocinfo: errno: *errno* **devno:** *devno*

X25DD entry_start: errno: *errno* **devno:** *devno* **cmd:** *cmd* **flag:** *flag* **chan:** *chan*

X25DD exit_start: errno: *errno* **devno:** *devno* **Suberror:** *suberror* **status:** *status* **session_id:** *sesson_id*

X25DD entry_query: errno: *errno* **devno:** *devno* **chan:** *chan*

X25DD exit_query: errno: *errno* **devno:** *devno* **status:** *status*

X25DD entry_reject_call: errno: *errno* **devno:** *devno* **cmd:** *cmd* **flag:** *flag* **chan:** *chan*

X25DD exit_reject_call: errno: *errno* **devno:** *devno* **chan:** *chan* **status:** *status* **session_id:** *sesson_id* **call_id:** *call_id*

X25DD entry_query_session: errno: *errno* **devno:** *devno* **flag:** *flag* **chan:** *chan*

X25DD exit_query_session: errno: *errno* **devno:** *devno* **chan:** *chan* **session_id:** *session_id*

X25DD entry_del_rid: errno: *errno* **devno:** *devno* **flag:** *flag* **chan:** *chan*

X25DD exit_del_rid: errno: *errno* **devno:** *devno* **router_id:** *router_id*

X25DD entry_query_rid: errno: *errno* **devno:** *devno* **flag:** *flag* **chan:** *chan*

X25DD exit_query_rid: errno: *errno* **devno:** *devno* **router_id:** *router_id*

X25DD entry_link_con: errno: *errno* **devno:** *devno* **chan:** *chan*

X25DD exit_link_con: errno: *errno* **devno:** *devno* **cmd:** *cmd* **chan:** *chan* **status:** *status*

X25DD entry_link_dis: errno: *errno* **devno:** *devno* **cmd:** *cmd* **flag:** *flag* **chan:** *chan*

X25DD exit_link_dis: errno: *errno* **devno:** *devno* **cmd:** *cmd* **chan:** *chan* **status:** *status*

X25DD entry_link_stat: errno: *errno* **devno:** *devno* **cmd:** *cmd* **flag:** *flag* **chan:** *chan*

X25DD exit_link_stat: errno: *errno* **devno:** *devno* **status:** *status* **packet:** *packet* **frame:** *frame* **physical:** *physical*

X25DD entry_local_busy: errno: *errno* **devno:** *devno*

X25DD exit_local_busy: errno: *errno* **devno:** *devno* **session_id:** *sesson_id* **busy_mode:** *busy_mode*

X25DD entry_counter_get: errno: *errno* **devno:** *devno* **flag:** *flag* **chan:** *chan*

X25DD exit_counter_get: errno: *errno* **devno:** *devno* **chan:** *chan* **counter_val:** *counter_val*

X25DD entry_counter_wait: errno: *errno* **devno:** *devno* **flag:** *flag* **chan:** *chan*

X25DD exit_counter_wait: errno: *errno* **devno:** *devno* **chan:** *chan* **counter_id:** *counter_id* **counter_num:** *counter_num*

X25DD entry_counter_read: errno: *errno* **devno:** *devno* **flag:** *flag* **chan:** *chan*

X25DD exit_counter_read: errno: *errno* **devno:** *devno* **chan:** *chan* **counter_id:** *counter_id* **counter_val:** *counter_val*

X25DD entry_counter_rem: errno: *errno* **devno:** *devno* **flag:** *flag* **chan:** *chan*

X25DD exit_counter_rem: errno: *errno* **devno:** *devno* **chan:** *chan* **counter_id:** *counter_id*

X25DD entry_diag_io: errno: *errno* **devno:** *devno* **cmd:** *cmd* **chan:** *chan*

X25DD exit_diag_io: errno: *errno* **devno:** *devno* **cmd:** *cmd* **chan:** *chan* **crd_rc:** *crd_rc*

X25DD entry_diag_mem: errno: *errno* **devno:** *devno* **cmd:** *cmd* **flag:** *flag* **chan:** *chan*

X25DD exit_diag_mem: errno: *errno* **devno:** *devno* **cmd:** *cmd* **chan:** *chan* **crd_rc:** *crd_rc*

X25DD exit_diag_card: errno: *errno* **devno:** *devno*

X25DD entry_diag_card: errno: *errno* **devno:** *devno*

X25DD entry_reset: errno: *errno* **devno:** *devno*

X25DD exit_reset: errno: *errno* **devno:** *devno*

X25DD entry_diag_task: errno: *errno* **devno:** *devno*

X25DD exit_diag_task: errno: *errno* **devno:** *devno*

X25DD entry_ucode_task: errno: *errno* **devno:** *devno*

X25DD exit_ucode_task: errno: *errno* **devno:** *devno*

X25DD entry_add_rid: errno: *errno* **devno:** *devno* **flag:** *flag* **chan:** *chan*

X25DD exit_add_rid: errno: *errno* **devno:** *devno* **router_id:** *router_id* **priority:** *priority* **action:** *action* **uid:** *uid*

X25DD entry_intr_stat: errno: *errno* **devno:** *devno*

X25DD exit_intr_stat: errno: *errno* **devno:** *devno*

X25DD entry_traceon: errno: *errno* **devno:** *devno*

X25DD exit_traceoff: errno: *errno* **devno:** *devno*

errno: *errno*	Error number
devno: *devno*	Major and minor device number
cmd: *cmd*	**ioctl** command
chan: *chan*	Channel number
flag: *flag*	Open mode
ext: *ext*	Pointer to extension data area
gp_rc: *gp_rc*	Internal return code (for reporting to service organization)
packet_type: *packet_type*	Type of X.25 packet being sent
session_id: *sesson_id*	Session identifier, created with the CIO_START ioctl
status: *status*	Status return code
events: *events*	Events mask passed to select
reventp: *reventp*	Events that were signalled by the select call
uiop: *uiop*	Pointer to the uio structure passed by a nonkernel user
channame: *channame*	Extension to the pathname on the open call
session_type: *session_type*	Session type, created with the CIO_START ioctl
block.code: *block.code*	Type of status block returned as described in the X.25 documentation
block.opt 0: *block.opt 0*	Type of status block returned as described in the X.25 documentation
block.opt 1: *block.opt 1*	Type of status block returned as described in the X.25 documentation
call_id: *call_id*	Incoming call identifier supplied to a listening session, used when creating the SVC_IN session type
router_id: *router_id*	Identifies the X.25 router table element
packet: *packet*	Status of the packet layer of the X.25 link
0	Disconnected
1	Connecting
2	Connected.
frame: *frame*	The status of the frame layer of the X.25 link
physical: *physical*	The status of the physical layer of the X.25 link
busy_mode: *busy_mode*	A flag controlling whether the driver goes in or out of local-busy mode, defined in the **X25/X25user.h** file
counter_val: *counter_val*	Value of the X.25 counter being referenced
counter_id: *counter_id*	Reference ID of the X.25 counter being referenced
counter_num: *counter_num*	Number of counters being waited on
crd_rc: *crd_rc*	Internal return code that can be reported to the service organization
priority: *priority*	Priority given to a router entry as documented in the X.25 documentation
action: *action*	Action given for a router entry as documented in the X.25 documentation

uid: *uid*
uid of the user submitting this router request

Suberror: *suberror*
Error starting an X.25 session:

None.

Device was not configured before OPEN.

Interrupt could not be registered.

Non-monitor START in monitor session.

Monitor START in non-monitor session.

START is not legal in D or R session.

START has an invalid session type.

226 : HKWD DD GIO

This event is recorded by the Graphics IO device driver.

Trace Hook IDs: 230 through 233

The following trace hook IDs are stored in the **/usr/include/sys/trchkid.h** file.

230: HKWD PTHREAD MUTEX LOCK

This event is recorded by the **pthread_mutex_lock** subroutine.

Recorded Data

pthread_mutex_lock lock_addr=*address* **lock=***status* **lock owner=***owner*

address
Address of the mutex lock

status
Possible values:

REQUESTED

IRST GOT

GOT

GOT after thread_tsleep

NOT GOT*owner*

User thread ID of the mutex lock.

231: HKWD PTHREAD MUTEX UNLOCK

This event is recorded by the **pthread_mutex_unlock** subroutine.

Recorded Data

pthread_mutex_unlock lock_addr=*address* **lock owner=***owner*

address	Address of the mutex lock
owner	User thread ID of the mutex lock.

232: HKWD PTHREAD SPIN LOCK

This event is recorded by the **pthread_spin_lock** internal subroutine.

Recorded Data

pthread_spin_lock lock_addr=*address* **lock=***status*

address	Address of the mutex lock
status	Possible values:

REQUESTED

FIRST GOT

GOT after thread_tsleep

NOT GOT

233: HKWD PTHREAD SPIN UNLOCK

This event is recorded by the **pthread_spin_unlock** internal subroutine.

Recorded Data

pthread_spin_unlock lock_addr=*address*

address	Address of the mutex lock

Trace Hook IDs: 240 through 252

The following trace hook IDs are stored in the **/usr/include/sys/trchkid.h** file.

240 : HKWD SYSX DLC START

This event is recorded by a Data Link Control (**/dev/dlcether**, **/dev/dlcsdlc**, or **/dev/dlctoken**) when an attachment to a remote station is started.

Recorded Data

LAN protocol physical LAN

attachment name station name

station address

LAN protocol	Type of LAN protocol:

Ethernetl

EEE_802.3

SDLC

Token_Ring

physical LAN Type of physical LAN:

EIA_RS232D

EIA_RS336

X_21

PC_Network_Broadband

Standard_Baseband_Ethernet

Smart_MODEM_Autodial

IEEE_802.3_Baseband_Ethernet

IEEE_802.4_Token_Bus

IEEE_802.5_Token_Ring

attachment name Name of attachment

station name Remote station name

station address
Remote station address.

241 : HKWD SYSX DLC HALT

This event is recorded by a Data Link Control (**/dev/dlcether**, **/dev/dlcsdlc**, or **/dev/dlctoken**) when an attachment to a remote station is halted.

Recorded Data

LAN protocol physical LAN

attachment name station name

station address

LAN protocol Type of LAN protocol:

Ethernet

IEEE_802.3

SDLC

Token_Ring

physical LAN Type of Physical LAN:

EIA_RS232D

EIA_RS336

X_21

PC_Network_Broadband

Standard_Baseband_Ethernet

Smart_MODEM_Autodial

IEEE_802.3_Baseband_Ethernet

IEEE_802.4_Token_Bus

IEEE_802.5_Token_Ring

attachment name Name of attachment
station name Remote station name

station address
Remote station address.

242 : HKWD SYSX DLC TIMER

This event is recorded by a Data Link Control (**/dev/dlcether**, **/dev/dlcsdlc**, or **/dev/dlctoken**) when an internal time expires.

Recorded Data

LAN protocol timer type

LAN protocol Type of LAN protocol:
Ethernet
IEEE_802.3
SDLC
Token_Ring
timer type Type_of_Timer:
Slow_Station_Poll
Idle_Station_Poll
Link_Station_Aborted
Link_Station_Receive_Inactivity
Command_Fail_Safe
Command_Repoll

I_Frame_Acknowledgement.

243 : HKWD SYSX DLC XMIT

This event is recorded by a Data Link Control (**/dev/dlcether**, **/dev/dlcsdlc**, or **/dev/dlctoken**) when a packet is sent.

Recorded Data

LAN protocol header data

LAN protocol Type of LAN protocol:
Ethernet
IEEE_802.3
SDLC
Token_Ring

header data
LAN header data.

244 : HKWD SYSX DLC RECV

This event is recorded by a Data Link Control (**/dev/dlcether**, **/dev/dlcsdlc**, or **/dev/dlctoken**) when a packet is received.

Recorded Data

LAN protocol header data

LAN protocol Type of LAN protocol
header data LAN header data.

245 : HKWD SYSX DLC PERF

This event is recorded by a Data Link Control (**/dev/dlcether**, **/dev/dlcsdlc**, or **/dev/dlctoken**) at key points in the Data Link Control program to record performance data. This trace hook will normally be used by the LAN Administrator during DLC debug.

Recorded Data

event LAN protocol

event Possible values:

Begin_Wait_Call

End_Wait_Call

Begin_Get_Rcv_Buffer

End_Get_Rcv_Buffer

Begin_HASH_Function

End_HASH_Function

Begin_Get_Transmit_Buffer

End_Get_Transmit_Buffer

Begin_Receive-Network_Data

Send_I_Frame_To_Device_Handler

Put_Write_Data_in_Xmit_Queue

Put_Write_XID_in_Xmit_Queue

T1_Timeout

T2_Timeout

T3_Timeout

Send_Start_to_Device_Handler

Receive_Discovery_Find_Command

Receive_Resolve_Find_Command

Open_Physical_Link

Device_Started

Send_Non_I_Frame_Data

Send_Datagram_Data

Send_Network_Data

T3_Abort_Timeout

LAN protocol Type of LAN protocol:

Ethernet

IEEE_802.3

SDLC

Token_Ring

246 : HKWD SYSX DLC MONITOR

This event is recorded by a Data Link Control (**/dev/dlcether**, **/dev/dlcsdlc**, or **/dev/dlctoken**) at key points in the Data Link Control program to record input commands, commands sent to the device handler, packets sent and packets received. This trace hook will normally be used by the LAN administrator during DLC debug.

Recorded Data

LAN Protocol LAN activity debug data

LAN protocol Type of LAN protocol:

Ethernet

IEEE_802.3

SDLC

Token_Ring

LAN activity Type of LAN activity:

Write_Command

Receive_Non_I_Data

Receive_I_Frame_Data

Input_Send_Command

Send_Command

Timer

Receive_Network_Data

debug data Debug data.

251 : HKWD NETERR

This hook ID records TCP/IP network error events. TCP/IP network error events are recorded by the network interface layer, most of which are return status codes from network adapter device drivers.

Recorded Data

Event:

NETERR CIO_OK ifp=*ifp*

NETERR CIO_BAD_MICROCODE ifp=*ifp*

NETERR CIO_BUF_OVFLW ifp=*ifp*

NETERR CIO_HARD_FAIL ifp=*ifp*

NETERR CIO_LOST_DATA ifp=*ifp*

NETERR CIO_NOMBUF ifp=*ifp*

NETERR CIO_NOT_STARTED ifp=*ifp*

NETERR CIO_TIMEOUT ifp=*ifp*

NETERR CIO_NET_RCVRY_ENTER ifp=*ifp*

NETERR CIO_NET_RCVRY_EXIT ifp=*ifp*

NETERR CIO_NET_RCVRY_MODE ifp=*ifp*

NETERR CIO_INV_CMD ifp=*ifp*

NETERR CIO_BAD_RANGE ifp=*ifp*

NETERR CIO_NETID_INV ifp=*ifp*

NETERR CIO_NETID_DUP ifp=*ifp*

NETERR CIO_NETID_FULL ifp=*ifp*

NETERR X25_BAD_CALL_ID ifp=*ifp*

NETERR X25_CLEAR ifp=*ifp*

NETERR X25_INV_CTR ifp=*ifp*

NETERR X25_NAME_USED ifp=*ifp*

NETERR X25_NOT_PVC ifp=*ifp*

NETERR X25_NO_ACK ifp=*ifp*

NETERR X25_NO_ACK_REQ ifp=*ifp*

NETERR X25_NO_LINK ifp=*ifp*

NETERR X25_NO_NAME ifp=*ifp*

NETERR X25_PROTOCOL ifp=*ifp*

NETERR X25_PVC_USED ifp=*ifp*

NETERR X25_RESET ifp=*ifp*

NETERR X25_TABLE ifp=*ifp*

NETERR X25_TOO_MANY_VCS ifp=*ifp*

NETERR X25_AUTH_LISTEN ifp=*ifp*

NETERR X25_BAD_PKT_TYPE ifp=*ifp*

NETERR X25_BAD_SESSION_TYPE ifp=*ifp*

NETERR invalid xmit complete intr ifp=*ifp*

NETERR if detach( ) fail ifp=*ifp*

NETERR find_input_type( ) fail ifp=*ifp*

NETERR no mbufs ifp=*ifp*

NETERR if not running ifp=*ifp*

NETERR clear indication ifp=*ifp*

NETERR unknown packet type ifp=*ifp*

NETERR NET_XMIT_FAIL ifp=*ifp*

NETERR NET_DETACH_FAIL ifp=*ifp*

NETERR ARP, wrong header ifp=*ifp*

NETERR ARP, unknown protocol ifp=*ifp*

NETERR ARP, ip broadcast address ifp=*ifp*

NETERR ARP, duplicate address ifp=*ifp*

NETERR ARP, arp table full ifp=*ifp*

ifp=*ifp* Address of network interface **if** structure.

252 : HKWD SYSC TCPIP

This hook ID records socket-type system call events. The socket layer records these events on entry and exit to socket-type subroutines.

Event:

SOCKET socket (*domain***, *type*, *protocol***)**

SOCKET bind (*s***, *name*, *namelen***)**

SOCKET listen (*s***, *backlog***)**

SOCKET accept (*s***, *addr*, *addrlen***)**

SOCKET connect (*s***, *name*, *namelen***)**

SOCKET socketpair (*d***, *type*, *protocol*, *sv***)**

SOCKET sendto (*s***, *msg*, *len*, *flags*, *to*, *tolen***)**

SOCKET send (*s***, *msg*, *len*, *flags***)**

SOCKET sendmsg (*s***, *msg*, *flags***)**

SOCKET recvfrom (*s***, *buf*, *len*, *flags*, *from*, *fromlen***)**

SOCKET recv (*s***, *buf*, *len*, *flags***)**

SOCKET recvmsg (*s***, *msg*, *flags***)**

SOCKET shutdown (*s***, *how***)**

SOCKET setsocketopt (*s***, *level*, *optname*, *optval*, *optlen***)**

SOCKET getsocketopt (*s***, *level*, *optname*, *optval*, *optlen***)**

SOCKET getsockname (*s***, *name*, *namelen***)**

SOCKET getpeername (*s***, *name*, *namelen***)**

SOCKET gethostid

SOCKET sethostid (*hostid***)**

SOCKET gethostname (*name***, *namelen*)**

SOCKET sethostname (*name***, *namelen*)**

SOCKET getdomainname (*name***, *namelen*)**

SOCKET setdomainname (*name***, *namelen*)**

domain	Specifies an address format (Internet or the operating system domain).
type	Specifies semantics of communication (for example, stream or datagram).
protocol	Specifies a particular protocol to be used with the socket.
s	Socket file descriptor.
name	Name that the socket will be bound to.
namelen	Length of the name.
backlog	Defines the maximum length for the queue of pending connections.
addr	Specifies the address of the connecting entry.
addrlen	Contains the amount of space pointed to by the *addr* parameter.
d	Specifies the domain.
sv	References new sockets.
msg	Points to the message that will be sent.
len	Specifies the length of the message.
flags	Specifies the options to be used in sending the message.
to	Specifies the address of the target.
tolen	Specifies the size of the target.
buf	Specifies the address where data is entered.
from	Specifies the source address.
fromlen	Initialized to the size of the buffer associated with the *from* parameter.
how	Determines the action of the shutdown is determined by the *how* parameter.
level	Specifies the level of the protocol (for example, socket or tcp).
optname	Passed uninterpreted to the appropriate protocol.
optval	Used to access option values.
optlen	Used to access option values.

hostid Integer identifying the host.

Trace Hook IDs: 253 through 25A

The following trace hook IDs are stored in the **/usr/include/sys/trchkid.h** file.

253 : HKWD SOCKET

This hook ID records TCP/IP socket layer events. TCP/IP socket layer events are recorded by socket-layer code, most of which records parameters passed to functions and return values from functions.

Recorded Data

Event:

socreate (*value***, *value*, *value*, *value*)**

sobind (*value***, *value*)**

solisten (*value***, *value***)**

sofree (*value***)**

soclose (*value***)**

return from soclose (*value***)**

soabort (*value***)**

soaccept (*value***, *value*, *value*, *value*)**

return from soaccept (*value***)**

soconnect (*value***, *value*)**

soconnect2 (*value***, *value*)**

soconnect2_out

sodisconnect (*value***)**

return from sodisconnect (*value***)**

sosend (*value***, *value*, *value*, *value*, *value*)**

return from sosend (*value***)**

soreceive (*value***, *value*, *value*, *value*, *value*)**

return from soreceive (*value***, *value*)**

soshutdown (*value***)**

sorflush (*value***, *value*, *value*, *value*)**

sosetopt (*value***)**

return from sosetopt (*value***, *value*, *value*, *value*)**

sogetopt (*value***, *value*, *value*, *value*)**

return from sogetopt

sohasoutofband (*value***)**

return from sohasoutofband

254 : HKWD MBUF

This hook word is used by the MBUF services routines to record **mbuf** activity. The **mbuf** routines are called by many system components. These routines record parameters passed to functions and the return values.

Recorded Data

Event:

m_get (*value***, *value***)**

return from m_get (*value***)**

m_getclr (*value***, *value***)**

return form m_getclr (*value***)**

m_free (*value***)**

return from m_free (*value***)**

m_copy (*value***, *value***, *value***)**

return from m_copy (*value***)**

m_copydata (*value***, *value***, *value***, *value***)**

return from m_copydata

m_pullup_1

m_pullup_2

mlowintr

return from mlowintr

m_low: schedule mlowintr

255 : HKWD IFEN

This hook ID is used by the Ethernet network interface to record interface events. The Ethernet network interface records packet transmit-and-receive operations and unusual interface conditions.

Recorded Data

Event:

en_statintr (entry) ifp=*ifp* **sbp_option=***sbp_option*

en_statintr (rtn)

en_netintr (entry) ifp=*ifp* **status=***status*

en_netintr (rtn)

en_attach (entry) unit=*unit*

en_attach (rtn)

en_detach (entry) ifp=*ifp*

en_detach (rtn)

en_init (entry)

en_init (rtn)

en_ioctl (entry) ifp=*ifp* **cmd=***cmd* **data=***data data*

en_ioctl (rtn) error=*error*

en_output (entry) ifp=*ifp* **m=***m* **family=***family* **dst_ipaddr=***dst_ipaddr*

en_output (rtn) error=*error*

en_reset (entry)

en_reset (rtn)

en_recv (entry) m=*m* **ifp=***ifp*

en_recv (rtn)

ifp=*ifp*	Address of network interface **if** structure
sbp_option=*sbp_option*	Status block option value
status=*status*	Status value
unit=*unit*	Network interface unit number
cmd=*cmd*	Value of ioctl command parameter
data=*data*	Value of ioctl data parameter
m=*m*	Address of **mbuf**
family=*family*	Address family value
dst_ipaddr=*dst_ipaddr*	Destination IP address value

error=*error*
Return status of interface output routine.

256 : HKWD IFTR

This hook ID is used by the token-ring network interface to record interface events. The token-ring network interface records packet transmit-and-receive operations and unusual interface conditions.

Recorded Data

Event:

ie5_statintr (entry) ifp=*ifp* **sbp_option=***sbp_option*

ie5_statintr (rtn)

ie5_netintr (entry) ifp=*ifp* **status=***status*

ie5_netintr (rtn)

ie5_attach (entry) unit=*unit*

ie5_attach (rtn)

ie5_detach (entry) ifp=*ifp*

ie5_detach (rtn)

ie5_init (entry)

ie5_init (rtn)

ie5_ioctl (entry) ifp=*ifp* **cmd=***cmd* **data=***data data*

ie5_ioctl (rtn) error=*error*

ie5_output (entry) ifp=*ifp* **m=***m* **family=***family* **dst_ipaddr=***dst_ipaddr*

ie5_output (rtn) error=*error*

ie5_reset (entry)

ie5_reset (rtn)

ie5_recv (entry) m=*m* **ifp=***ifp*

ie5_recv (rtn)

ifp=*ifp*	Address of network interface **if** structure
sbp_option=*sbp_option*	Status block option value
status=*status*	Status value
unit=*unit*	Network interface unit number
cmd=*cmd*	Value of ioctl command parameter
data=*data*	Value of ioctl data parameter
m=*m*	Address of **mbuf**
family=*family*	Address family value
dst_ipaddr=*dst_ipaddr*	Destination IP address value

error=*error*
Return status of interface output routine.

257 : HKWD IFET

This hook ID is used by the 802.3 network interface to record interface events. The 802.3 network interface records packet transmit-and-receive operations and unusual interface conditions.

Recorded Data

Event:

ie3_statintr (entry) ifp=*ifp* **sbp_option=***sbp_option*

ie3_statintr (rtn)

ie3_netintr (entry) ifp=*ifp* **status=***status*

ie3_netintr (rtn)

ie3_attach (entry) unit=*unit*

ie3_attach (rtn)

ie3_detach (entry) ifp=*ifp*

ie3_detach (rtn)

ie3_init (entry)

ie3_init (rtn)

ie3_ioctl (entry) ifp=*ifp* **cmd=***cmd* **data=***data data*

ie3_ioctl (rtn) error=*error*

ie3_output (entry) ifp=*ifp* **m=***m* **family=***family* **dst_ipaddr=***dst_ipaddr*

ie3_output (rtn) error=*error*

ie3_reset (entry)

ie3_reset (rtn)

ie3_recv (entry) m=*m* **ifp=***ifp*

ie3_recv (rtn)

ifp=*ifp*	Address of network interface **if** structure
sbp_option=*sbp_option*	Status block option value
status=*status*	Status value
unit=*unit*	Network interface unit number
cmd=*cmd*	Value of ioctl command parameter
data=*data*	Value of ioctl data parameter
m=*m*	Address of **mbuf**
family=*family*	Address family value
dst_ipaddr=*dst_ipaddr*	Destination IP address value

error=*error*
Return status of interface output routine.

258 : HKWD IFXT

This hook ID is used by the X.25 network interface to record interface events. The X.25 network interface records packet transmit-and-receive operations and unusual interface conditions.

Recorded Data

Event:

xt_statintr (entry) ifp=*ifp* **sbp_option=***sbp_option*

xt_statintr (rtn)

xt_netintr (entry) ifp=*ifp* **status=***status*

xt_netintr (rtn)

xt_attach (entry) unit=*unit*

xt_attach (rtn)

xt_detach (entry) ifp=*ifp*

xt_detach (rtn)

xt_init (entry)

xt_init (rtn)

xt_ioctl (entry) ifp=*ifp* **cmd=***cmd* **data=***data data*

xt_ioctl (rtn) error=*error*

xt_output (entry) ifp=*ifp* **m=***m* **family=***family* **dst_ipaddr=***dst_ipaddr*

xt_output (rtn) error=*error*

xt_reset (entry)

xt_reset (rtn)

xt_recv (entry) m=*m* **ifp=***ifp*

xt_recv (rtn)

ifp=*ifp*	Address of network interface **if** structure
sbp_option=*sbp_option*	Status block option value
status=*status*	Status value
unit=*unit*	Network interface unit number
cmd=*cmd*	Value of ioctl command parameter
data=*data*	Value of ioctl data parameter
m=*m*	Address of **mbuf**
family=*family*	Address family value
dst_ipaddr=*dst_ipaddr*	Destination IP address value

error=*error*
Return status of interface output routine.

259 : HKWD IFSL

This hook ID is used by the SLIP network interface to record interface events. The SLIP network interface records packet transmit and receive operations and unusual interface conditions.

Recorded Data

Event:

slattach (entry) unit=*unit*

slattach (rtn)

sl_detach (entry) ifp=*ifp*

sl_detach (rtn)

slinit (entry)

slinit (rtn)

slioctl (entry) ifp=*ifp* **cmd=***cmd* **data=***data*

slioctl (rtn) error=*error*

sloutput (entry) ifp=*ifp* **m=***m* **family=***family* **dst_ipaddr=***dst_ipaddr*

sloutput (rtn) error=*error*

slreset (entry)

slreset (rtn)

unit=*unit* Network interface unit number
ifp=*ifp* Address of network interface **if** structure
cmd=*cmd* Value of ioctl command parameter
data=*data* Value of ioctl data parameter
m=*m* Address of **mbuf**
family=*family* Address family value

dst_ipaddr=*dst_ipaddr*
Destination IP address value

error=*error*
Return status of interface output routine.

25A : HKWD TCPDBG

This event ID is used to trace TCP events. The TCP protocol records outgoing and incoming packet events when the socket used has had the SO_DEBUG option turned on for the socket.

Recorded Data

Events:

TA_INPUT tp=*tp* **ostate=***ostate* **flags=***flags*

TA_OUTPUT tp=*tp* **ostate=***ostate* **flags=***flags*

TA_USER req=*req*

TA_RESPOND tp=*tp* **ostate=***ostate* **flags=***flags*

TA_DROP tp=*tp* **ostate=***ostate* **flags=***flags*

seq=*seq* **ack=***ack* **len=***len*

rcvnxt=*rcvnxt* **rcvwnd=***rcvwnd* **snduna=***snduna*

sndmax=*sndmax*

sndw11=*sndw11* **sndw12=***sndw12* **sndwnd=***sndwnd*

tp=*tp*	Address control block structure
ostate=*ostate*	State of tcp connection
flags=*flags*	Flags value for incoming/outgoing packet
req=*req*	Type of user request
seq=*seq*	Sequence number value
ack=*ack*	ack value
len=*len*	Length of data
rcvnxt=*rcvnxt*	Receive next value
rcvwnd=*rcvwnd*	Receive window value
snduna=*snduna*	Send unnumbered acknowledgement value
sndmax=*sndmax*	Send maximum value
sndw11=*sndw11*	Send w11 value
sndw12=*sndw12*	Send w12 value

sndwnd=*sndwnd*
Send window value.

Trace Hook IDs: 271 through 280

The following trace hook IDs are stored in the **/usr/include/sys/trchkid.h** file.

271: HKWD SNA API

This error is logged by the SNA Services upon entry and exit of the SNA API command routines.

Recorded Data

Event:

SNA API Commands Entry SNA_API Open Connection ID=*cid* **Resource ID=***rid* **Buffer Address=***buf* **Buffer Length=***len*

SNA API Commands Exit SNA_API Open Connection ID=*cid* **Resource ID=***rid* **Buffer Address=***buf* **Return Code=***rc*

SNA API Commands Entry SNA_API Close Connection ID=*cid* **Resource ID=***rid* **Buffer Address=***buf* **Buffer Length=***len*

SNA API Commands Exit SNA_API Close Connection ID=*cid* **Resource ID=***rid* **Buffer Address=***buf* **Return Code=***rc*

SNA API Commands Entry SNA_API IOCTL Connection ID=*cid* **Resource ID=***rid* **Buffer Address=***buf* **Buffer Length=***len* **Request=***ioctl req*

SNA API Commands Exit SNA_API IOCTL Connection ID=*cid* **Resource ID=***rid* **Buffer Address=***buf* **Return Code=***rc* **Request=***ioctl req*

SNA API Commands Entry SNA_API Write Connection ID=*cid* **Resource ID=***rid* **Buffer Address=***buf* **Buffer Length=***len*

SNA API Commands Exit SNA_API Write Connection ID=*cid* **Resource ID=***rid* **Buffer Address=***buf* **Return Code=***rc*

SNA API Commands Entry SNA_API Read Connection ID=*cid* **Resource ID=***rid* **Buffer Address=***buf* **Buffer Length=***len*

SNA API Commands Exit SNA_API Read Connection ID=*cid* **Resource ID=***rid* **Buffer Address=***buf* **Return Code=***rc*

SNA API Commands Entry SNA_API MPX Connection ID=*cid* **Resource ID=***rid* **Buffer Address=***buf* **Buffer Length=***len*

SNA API Commands Exit SNA_API MPX Connection ID=*cid* **Resource ID=***rid* **Buffer Address=***buf* **Return Code=***rc*

SNA API Commands Entry SNA_API Select Connection ID=*cid* **Resource ID=***rid* **Buffer Address=***buf* **Buffer Length=***len* **Request=***select req*

SNA API Commands Exit SNA_API Select Connection ID=*cid* **Resource ID=***rid* **Buffer Address=***buf* **Return Code=***rc* **Request=***select req*

SNA API Commands Entry SNA_API Config Connection ID=*cid* **Resource ID=***rid* **Buffer Address=***buf* **Buffer Length=***len* **Request=***config req*

SNA API Commands Exit SNA_API Config Connection ID=*cid* **Resource ID=***rid* **Buffer Address=***buf* **Return Code=***rc* **Request=***config req*

Connection ID=*cid*	Connection identifier
Resource ID=*rid*	Resource identifier
Buffer Address=*buf*	Buffer address
Buffer Length=*len*	Buffer length
Return Code=*rc*	SNA return code as defined in the **luxsna.h** file
Request=*icoctl req*	ioctl operation:

Allocate

Deallocate

Confirm

Confirmed

Flush

Prepare_To_Receive

Request_To_Send

Send_FMH

Send_Error

Get_Attribute

Send_Status

Get_Status

CP_Status

Allocate_Listen

Get_Parameters

Request=*select req* Select operation:

Asynchronous - Read

Asynchronous - Write

Asynchronous - Write,Read

Asynchronous - Exception

Asynchronous - Exception,Read

Asynchronous - Exception,Write

Asynchronous - Exception,Write,Read

Synchronous - Read

Synchronous - Write

Synchronous - Write,Read

Synchronous - Exception

Synchronous - Exception,Read

Synchronous - Exception,Write

Synchronous - Exception,Write,Read

Request=*config req* Configuration operation:

Initiate

Terminate

Query

280: HKWD HIA

This error is logged by the HIA device driver.

Recorded Data

Event:

HIADD Ccls Entry to device close routine:
d1=*d1* Device minor number
d2=*d2* Session address number

d3=*d3* Pointer to close extension structure, if any.

HIADD CclE Exit from device close routine:
d1=*d1* Status of link.

HIADD linS Entry to top half of device head interrupt handler routine:
d1=*d1* Session address number
d2=*d2* Operation results passed up from device handler
d3=*d3* Interrupt type passed up from device handler.

HIADD linE Exit to top half of device head interrupt handler routine:

d1=d*1*

Session address number.

HIADD lioS

Entry to **ioctl** routine:

d1=*d1*

Device minor number

d2=*d2*

ioctl command parameter

d3=*d3*

Pointer to ioctl arg parameter

d4=*d4*

Value of ioctl flag parameter.

HIADD lio1 Second trace point for start of ioctl entry point:

d1=*d1*

Session address number.

HIADD lioE Exit of **ioctl** routine:

d1=*d1*

Link address status.

HIADD MpxS Entry to **mpx** routine:

d1=*d1*

Device minor number

d2=*d2*

Session address number

d3=*d3*

First character of channel name

d4=*d4*

State of the DDS.

HIADD MpxE Exit **mpx** routine:

d1=*d1*

Device minor number

d2=*d2*

Address of session address number

d3=*d3*

Address of channel name string

d4=*d4*

Session address number.

HIADD OpeS Entry to **open** routine:

d1=*d1*

Device minor session

d2=*d2*

Read/write flag

d3=*d3*

Session address number

d4=*d4*

Address of DDS.

HIADD OpeE Exit **open** routine:

d1=*d1*

Address of DDS

d2=*d2*

Session address number.

HIADD RrdS Entry to **read** routine:

d1=*d1*

State of DDS

d2=*d2*

Session address number

d3=*d3*

Address of **ext** structure, used with the **readx** subroutine.

HIADD RrdE Exit **read** routine:

d1=*d1*

Value entry point will return

d2=*d2*

Value of link address IO flag

d3=*d3*

Status of link address IO.

HIADD SslS Entry to **select** routine:

d1=*d1*

Device number

d2=*d2*

Events to select on

d3=*d3*

Session address number.

HIADD SslE Exit **select** routine:

d1=*d1*

Device number

d2=*d2*

Events to select on

d3=*d3*

Status of events selected on

d4=*d4*

Session address number.

HIADD WwrS Entry to **write** routine:

d1=*d1*

State of DDS

d2=*d2*

Session address number

d3=*d3*

Address of **ext** structure, used with the **writex** subroutine.

HIADD WwrE Exit **write** routine:

d1=*d1*

Status of link

d2=*d2*

Value of link address IO flag

d3=*d3*

Status of link address IO

d4=*d4*

Return value for entry point.

HIADD CDDs Entry for **hia** configuration:

d1=*d1*

Device number

d2=*d2*

Configuration command

d3=*d3*

Flag indicating if this is first open.

HIADD CDDe Exit **hia** configuration routine:

d1=*d1*

Return value.

HIADD INTO Device handler I/O routine:

d1=*d1*

Interrupt status.

HIADD INT2 Beginning of device handler I/O routine:

d1=*d1*

stb

d2=*d2*

icc

d3=*d3*

ccb

d4=*d4*

lda.

HIADD INT3 Beginning of device handler I/O routine:

d1=*d1*

count

d2=*d2*

ipf

d3=*d3*

vda[0]

d4=*d4*

vda[1].

HIADD INTz Beginning of device handler I/O routine:

d1=*d1*

xrc.

HIADD INT6 Beginning of device handler I/O routine:

d1=*d1*

Type of IO requested

d2=*d2*

Address of link address structure.

HIADD INT9 Unrecognized interrupt.

HIADD IIOs Entry to I/O handler portion of bottom half:

d1=*d1*

Session number

d2=*d2*

Type of IO requested.

HIADD IIOe Exit to I/O handler portion of bottom half:

d1=*d1*

Return value.

HIADD RIO0 Entry to routine to update the romp to **hia** area to send the **hia** a new command:

d1=*d1*

Interrupt pending flag value

d2=*d2*

Command control byte

d3=*d3*

Flag byte

d4=*d4*

Minor session number of the link address.

HIADD RIO1 Entry to routine to update the romp to **hia** area to send the **hia** a new command:

d1=*d1*

Size of the data to transfer

d2=*d2*

Address of the buffer to transfer

d3=*d3*

Variable data area.

HIADD RIO2 Routine to update the romp to **hia** area to send the **hia** a new command:

d1=*d1*

First byte of the buffer

d2=*d2*

In-use flag of the buffer

d3=*d3*

Count of data for transfer

d4=*d4*

Offset of data in buffer.

HIADD RIO3 Routine to update the romp to **hia** area to send the **hia** a new command:

d1=*d1*

dma base

d2=*d2*

dma channel ID

d3=*d3*

dma memory block.

HIADD RIO4 Routine to update the romp to **hia** area to send the **hia** a new command:

d1=*d1*

First byte of data

d2=*d2*

Second byte of data

d3=*d3*

Third byte of data

d4=*d4*

Fourth byte of data.

HIADD RIO5 End of routine to update the romp to **hia** area to send the **hia** a new command.

HIADD SOFs Entry to the main off-level routine:

d1=*d1*

Type of interrupt processing, should be 70 to indicate off-level

d2=*d2*

Address of DDS.

HIADD SOFe Exit from the main off-level routine.

HIADD YOF1 Entry to routine to handle interrupt to indicate statistical data has been reported by the **hia**:

d1=*d1*

Count

d2=*d2*

Session address number

d3=*d3*

Status

d4=*d4*

Address of read buffer.

HIADD stmr Routine to set timers:

d1=*d1*

Timer ID

d2=*d2*

Time count.

HIADD utmr Routine to cancel timers:

d1=*d1*

Timer ID.

Trace Hook IDs: 301 through 315

The following trace hook IDs are stored in the **/usr/include/sys/trchkid.h** file.

301: HKWD KERN ASSERTWAIT

This event is recorded by the **e_assert_wait** kernel service.

Recorded Data

e_assert_wait: tid=*tid* **anchor=***anchor* **flag=***flag* **lr=***lr*

tid	Thread ID of the calling kernel thread.
anchor	The *event_word* parameter; the anchor to the list of kernel threads waiting on this event.
flag	The *interruptible* parameter.
lr	Value of the link register, specifying the return address of the service.

302: HKWD KERN CLEARWAIT

This event is recorded by the **e_clear_wait** kernel service.

Recorded Data

e_clear_wait: tid=*tid* **anchor=***anchor* **result=***result* **lr=***lr*

tid	The *tid* parameter; the thread ID of the kernel thread to be awakened.
anchor	Anchor to the event list where the target thread is sleeping.
result	The *result* parameter; the value to return to the awkened thread.
lr	Value of the link register, specifying the return address of the service.

303: HKWD KERN THREADBLOCK

This event is recorded by the **e_block_thread** kernel service.

Recorded Data

e_block_thread: tid=*tid* **anchor=***anchor* **t_flags=***t_flags* **lr=***lr*

tid	Thread ID of the calling kernel thread.
anchor	Anchor to the event list where the kernel thread will sleep.
t_flags	Flags of the kernel thread.
lr	Value of the link register, specifying the return address of the service.

304: HKWD KERN EMPSLEEP

This event is recorded by the **e_mpsleep** kernel service.

Recorded Data

e_mpsleep: tid=*tid* **anchor=***anchor* **timeout=***timeout* **lock=***lock* **flags=***flags* **lr=***lr*

tid	Thread ID of the calling kernel thread.
anchor	The *event_word* parameter; the anchor to the list of kernel threads waiting on this event.
timeout	The *timeout* parameter; the timeout for the sleep.
lock	The *lock_word* parameter; the lock (simple or complex) to unlock by the kernel service.
flags	The *flags* parameter; the lock and signal handling options.
lr	Value of the link register, specifying the return address of the service.

305: HKWD KERN EWAKEUPONE

This event is recorded by the **e_wakeup_one** kernel service.

Recorded Data

e_wakeup_one: tid=*tid* **anchor=***anchor* **lr=***lr*

tid	Thread ID of the calling kernel thread.
anchor	The *event_word* parameter; the anchor to the list of kernel threads waiting on this event.

lr	Value of the link register, specifying the return address of the service.

306: HKWD SYSC CRTHREAD

This event is recorded by the **thread_create** system call.

Recorded Data

thread_create: pid=*pid* **tid=***tid* **priority=***priority* **policy=***policy*

pid	Process ID of the calling kernel thread's process.
tid	Thread ID of the calling kernel thread.
priority	Priority of the new kernel thread.
policy	Scheduling policy of the new kernel thread.

307: HKWD KERN KTHREADSTART

This event is recorded by the **kthread_start** kernel service.

Recorded Data

kthread_start: pid=*pid* **tid=***tid* **priority=***priority* **policy=***policy* **func=***func*

pid	Process ID of the calling kernel thread's process.
tid	The *tid* parameter; the thread ID of the kernel thread to start.
priority	Priority of the new kernel thread.
policy	Scheduling policy of the new kernel thread.
func	The *i_func* parameter, the address of the new kernel thread's entry-point routine.

308 : HKWD SYSC TERMTHREAD

This event is recorded by the **thread_terminate** system call.

Recorded Data

thread_terminate: pid=*pid* **tid=***tid*

pid	Process ID of the calling kernel thread's process.
tid	Thread ID of the calling kernel thread.

309 : HKWD KERN KSUSPEND

This event is recorded by the **ksuspend** subroutine. This subroutine is used internally by the system and is undocumented.

Recorded Data

ksuspend: tid=*tid* **p_suspended=***suspended* **p_active=***active*

tid	Thread ID of the calling kernel thread.
suspended	Number of suspended kernel threads in the process.
active	Number of active (suspendable) kernel threads in the process.

310 : HKWD SYSC THREADSETSTATE

This event is recorded by the **thread_setstate** system call.

Recorded Data

thread_setstate: tid=*tid* **t_state=***t_state* **t_flags=***t_flags* **priority=***priority* **policy=***policy*

tid	Thread ID of the target kernel thread.
t_state	Current state of the kernel thread. Possible values: **NONE** **IDLE** **RUN** **SLEEP** **SWAP** **STOP** **ZOMB**
t_flags	New flags of the kernel thread.
priority	New priority of the kernel thread.
policy	New scheduling policy of the kernel thread.

311 : HKWD SYSC THREADTERM ACK

This event is recorded by the **thread_terminate_ack** system call.

Recorded Data

thread_terminate_ack: current_tid=*crt_tid* **target_tid=***targ_tid*

crt_tid	Thread ID of the calling kernel thread.
targ_tid	Thread ID of the target kernel thread.

312 : HKWD SYSC THREADSETSCHED

This event is recorded by the **thread_setsched** system call.

Recorded Data

thread_setsched: pid=*pid* **tid=***tid* **priority=***priority* **policy=***policy*

pid	Process ID of the calling kernel thread's process.
tid	The *tid* parameter; the thread ID of the target kernel thread.
priority	The *priority* parameter; the priority to set.
policy	The *policy* parameter; the scheduling policy to set.

313 : HKWD KERN TIDSIG

This event is recorded by the **tidsig** subroutine. This subroutine is used internally by the system and is undocumented.

Recorded Data

tidsig: pid=*pid* **tid=***tid* **signal=***signal* **lr=***lr*

pid	Process ID of the calling kernel thread's process.

tid	Thread ID of the calling kernel thread.
signal	Symbolic name of the delivered signal.
lr	Value of the link register, specifying the return address of the routine.

314 : HKWD KERN WAITLOCK

This event is recorded by the **wait_on_lock** subroutine. This subroutine is used internally by the system and is undocumented.

Recorded Data

wait_on_lock: pid=*pid* **tid=***tid* **lockaddr=***lockaddr*

pid	Process ID of the calling kernel thread's process.
tid	Thread ID of the calling kernel thread.
lockaddr	Address of the lock.

315 : HKWD KERN WAKEUPLOCK

This event is recorded by the **wakeup_lock** subroutine. This subroutine is used internally by the system and is undocumented.

Recorded Data

wakeup_lock: lockaddr=*lockaddr* **waiters=***waiters*

lockaddr	Address of the lock.
waiters	Number of kernel threads remaining sleeping on the lock.

Trace Hook IDs: 3C5 through 3E2

3c5 : HKWD SYSC IPCACCESS

This event is recorded by the **msgctl**, **msgrcv**, **msgsnd**, **semctl**, **semop**, **shmat**, and **shmctl** subroutines.

Recorded Data

ipcaccess p->uid=*value* **p->mode**=*value* **p->seq**=*value* **p->key**=*value* **mode**=*value*

p->uid=*value*	The user id of the ipc object creator.
p->mode=*value*	The mode of the ipc object.
p->seq=*value*	The slot usage sequence number of the ipc object.
p->key=*value*	The key of the ipc object.
mode=*value*	The mode being requested.

3c6 : HKWD SYSC IPCGET

This event is recorded by the **msgget**, **semget** and **shmget** subroutines.

Recorded Data

ipcget key=*value* **flag**=*value* **base**=*value* **size**=*value* ***mark**=*value*

key=*value*	The key of the requested ipc object.

flag=*value* The get flags.
base=*value* The base address of the ipc object array.
size=*value* The size of each ipc object.
***mark**=*value* The largest used index into the ipc object array.

3c7 : HKWD SYSC MSGCONV

This event is recorded by the **msgctl**, **msgrcv**, **msgsnd** and **msgselect** subroutines.

Recorded Data

msgconv msgid=*value* **seq**=*value* **index**=*value* **qp**=*value*

msgid=*value* The id of the message queue.
seq=*value* The slot usage sequence number of the message queue.
index=*value* The index into the message queue array.
qp=*value* The pointer to the message queue.

3c8 : HKWD SYSC MSGCTL

This event is recorded by the **msgctl** subroutine.

Recorded Data

msgctl msgid=*value* **cmd**=*value* **buf**=*value*

msgid=*value* The id of the message queue.
cmd=*value* The command to perform.
buf=*value* The buffer used by the command.

3c9 : HKWD SYSC MSGGET

This event is recorded by the **msgget** subroutine.

Recorded Data

msgget key=*value* **msgflg**=*value* **msgid**=*value* **rval**=*value*

key=*value* The key of the requested message queue.
msgflg=*value* The get flags.
msgid=*value* The id of the message queue.
rval=*value* The pointer to the message queue.

3ca : HKWD SYSC MSGRCV

This event is recorded by the **msgrcv** subroutine.

Recorded Data

msgrcv msgid=*value* **msgp**=*value* **msgsz**=*value* **msgtyp**=*value* **msgflg**=*value*

msgid=*value* The id of the message queue.
msgp=*value* The pointer to the message buffer.
msgsz=*value* The size of the message.
msgtyp=*value* The type of the message.

msgflg=*value* The receive flags.

3cb : HKWD SYSC MSGSELECT

This event is recorded by the **msgselect** subroutine.

Recorded Data

msgselect msgid=*value* **corl**=*value* **reqevents**=*value* **rtneventsp**=*value*

msgid=*value* The id of the message queue.
corl=*value* The correlator of the select.
reqevents=*value* The requested events
rtneventsp=*value* The buffer for recorded events.

3cc : HKWD SYSC MSGSND

This event is recorded by the **msgsnd** subroutine.

Recorded Data

msgsnd msgid=*value* **msgp**=*value* **msgsz**=*value* **msgflg**=*value*

msgid=*value* The id of the message queue.
msgp=*value* The pointer to the message buffer.
msgsz=*value* The size of the message.
msgflg=*value* The send flags.

3cd : HKWD SYSC MSGXRCV

This event is recorded by the **msgxrcv** subroutine.

Recorded Data

msgxrcv msgid=*value* **msgp**=*value* **msgsz**=*value* **msgtyp**=*value* **msgflg**=*value*

msgid=*value* The id of the message queue.
msgp=*value* The pointer to the message buffer.
msgsz=*value* The size of the message.
msgtyp=*value* The type of the message.
msgflg=*value* The receive flags.

3ce : HKWD SYSC SEMCONV

This event is recorded by the **semctl**, **exit** and **semop** subroutines.

Recorded Data

semconv semid=*value* **seq**=*value* **index**=*value* **sp**=*value*

semid=*value* The id of the semaphore set.
seq=*value* The slot usage sequence number of the message queue.
index=*value* The index into the semaphore set array.
sp=*value* The pointer to the semaphore set.

3cf : HKWD SYSC SEMCTL

This event is recorded by the **semctl** subroutine.

Recorded Data

semctl semid=*value* **semnum**=*value* **cmd**=*value* **arg**=*value*

semid=*value*	The id of the semaphore set.
semnum=*value*	The number of the semaphore in the set.
cmd=*value*	The command to perform.
arg=*value*	The argument to the command.

3d0 : HKWD SYSC SEMGET

This event is recorded by the **semget** subroutine.

Recorded Data

semget key=*value* **nsems**=*value* **semflg**=*value* **sp**=*value*

key=*value*	The key of the requested semaphore set.
nsems=*value*	The number of semaphores requested.
semflg=*value*	The get flags.
sp=*value*	Pointer to the semaphore set.

3d1 : HKWD SYSC SEMOP

This event is recorded by the **semop** subroutine.

Recorded Data

semop semid=*value* **sops**=*value* **nsops**=*value*

semid=*value*	The id of the semaphore set.
sops=*value*	The semaphore operations.
nsops=*value*	The number of semaphore operations.

3d2 : HKWD SYSC SEM

This event is recorded by the **semop** subroutine.

Recorded Data

semop semid=*value* **semval**=*value* **sem_num**=*value* **sem_op**=*value* **sem_flg**=*value*

semid=*value*	The id of the semaphore set.
semval=*value*	The current semaphore value.
sem_num=*value*	The semaphore number.
sem_op=*value*	The semaphore operation.
sem_flg=*value*	The operation flags.

3d3 : HKWD SYSC SHMAT

This event is recorded by the **shmat** subroutine.

Recorded Data

shmat shmid=*value* **addr**=*value* **flag**=*value*

shmid=*value*	The id of the shared memory region.
addr=*value*	The address to attach to.
flag=*value*	The attach flags.

3d4 : HKWD SYSC SHMCONV

This event is recorded by the **shmat** and **shmctl** subroutines.

Recorded Data

shmconv shmid=*value* **flg**=*value* **seq**=*value* **index**=*value* **sp**=*value*

shmid=*value*	The id of the shared memory region.
flg=*value*	The operation flags.
seq=*value*	The slot usage sequence number of the shared memory region.
index=*value*	The index into the shared memory region array.
sp=*value*	The pointer to the shared memory region.

3d5 : HKWD SYSC SHMCTL

This event is recorded by the **shmctl** subroutine.

Recorded Data

shmctl shmid=*value* **cmd**=*value* **arg**=*value*

shmid=*value*	The id of the shared memory region.
cmd=*value*	The command to perform.
arg=*value*	The argument to the command.

3d6 : HKWD SYSC SHMDT

This event is recorded by the **shmdt** subroutine.

Recorded Data

shmdt addr=*value*

addr=*value*	The address to detach from.

3d7 : HKWD SYSC SHMGET

This event is recorded by the **shmget** subroutine.

Recorded Data

shmget key=*value* **size**=*value* **shmflg**=*value* **sp**=*value*

key=*value*	The id of the shared memory region.
size=*value*	The size of the shared memory region.
shmflg=*value*	The get flags.
sp=*value*	The pointer to the shared memory region.

3d8 : HKWD SYSC MADVISE

This event is recorded by the **madvise** subroutine.

Recorded Data

madvise addr=*value* **len**=*value* **behav**=*value*

addr=*value*	The address to advise on.
len=*value*	The length of the region to advise on.
behav=*value*	The behavior to advise.

3d9 : HKWD SYSC MINCORE

This event is recorded by the **mincore** subroutine.

Recorded Data

mincore addr=*value* **len**=*value* **vec**=*value*

addr=*value*	The address to check.
len=*value*	The length of the region to check.
vec=*value*	The state of the pages.

3da : HKWD SYSC MMAP

This event is recorded by the **mmap** subroutine.

Recorded Data

mmap addr=*value* **len**=*value* **prot**=*value* **flags**=*value* **fd**=*value*

addr=*value*	The address to map to.
len=*value*	The length of the region to map.
prot=*value*	The protection of the region to map.
flags=*value*	The map flags.
fd=*value*	The file descriptor to map.

3db : HKWD SYSC MPROTECT

This event is recorded by the **mprotect** subroutine.

Recorded Data

mprotect addr=*value* **len**=*value* **prot**=*value*

addr=*value*	The address to protect.
len=*value*	The length of the region to protect.
prot=*value*	The protection requested.

3dc : HKWD SYSC MSYNC

This event is recorded by the **msync** subroutine.

Recorded Data

msync addr=*value* **len**=*value*

addr=*value*	The address to sync.
len=*value*	The length of the region to sync.

3dd : HKWD SYSC MUNMAP

This event is recorded by the **munmap** subroutine.

Recorded Data

munmap addr=*value* **len**=*value*

addr=*value*	The address to unmap.
len=*value*	The length of the region to unmap.

3de : HKWD SYSC MVALID

This event is recorded by the **mvalid** subroutine.

Recorded Data

mvalid addr=*value* **len**=*value* **prot**=*value*

addr=*value*	The address to validate.
len=*value*	The length of the region to validate.
prot=*value*	The protection requested.

3df : HKWD SYSC MSEM_INIT

This event is recorded by the **msem_init** subroutine.

Recorded Data

msem_init msem=*value* **msem_state**=*value* **msem_wanted**=*value* **initial_value**=*value*

msem=*value*	The pointer to the msemaphore.
msem_state=*value*	The state of the msemaphore after.
msem_wanted=*value*	Threads waiting on the msemaphore.
initial_value=*value*	The initial value of the msemaphore

3e0 : HKWD SYSC MSEM_LOCK

This event is recorded by the **msem_lock** subroutine.

Recorded Data

msem_lock msem=*value* **msem_state**=*value* **msem_wanted**=*value* **condition**=*value*

msem=*value*	The pointer to the msemaphore.
msem_state=*value*	The current state of the msemaphore.
msem_wanted=*value*	The threads waiting on the msemaphore.
condition=*value*	The flags for the operation.

3e1 : HKWD SYSC MSEM_REMOVE

This event is recorded by the **msem_remove** subroutine.

Recorded Data

msem_remove msem=*value* **msem_state**=*value* **msem_wanted**=*value*

msem=*value*	The pointer to the msemaphore.
msem_state=*value*	The current state of the msemaphore.
msem_wanted=*value*	The threads waiting on the msemaphore.

3e2 : HKWD SYSC MSEM_UNLOCK

This event is recorded by the **msem_unlock** subroutine.

Recorded Data

msem_unlock msem=*value* **msem_state**=*value* **msem_wanted**=*value* **condition**=*value*

msem=*value*	The pointer to the msemaphore.
msem_state=*value*	The current state of the msemaphore.
msem_wanted=*value*	The threads waiting on the msemaphore.
condition=*value*	The flags for the operation.

Trace Hook IDs: 401

The following trace hook IDs are stored in the **/usr/include/sys/trchkid.h** file.

401 : HKWD TTY TTY

This event is recorded by the TTY device driver.

Recorded Data

Event:

(*maj***,** *min***,** *chan***) tty config cmd** *cmd* **ret** *ret*

(*maj***,** *min***,** *chan***) tty alloc cin** *cin* **cmd** *alloc cmd* **ret** *ret*

(*maj***,** *min***,** *chan***) tty open mode** *open mode* **ext** *ext* **ret** *ret*

(*maj***,** *min***,** *chan***) tty close ret** *ret*

(*maj***,** *min***,** *chan***) tty read ret** *ret*

(*maj***,** *min***,** *chan***) tty write ret** *ret*

(*maj***, *min*, *chan*) tty ioctl cmd** *ioctl cmd* **arg** *ioctl arg* **mode** *mode* **ret** *ret*

(*maj***, *min*, *chan*) tty select events** *events* **revents** *revents* **ret** *ret*

(*maj***, *min*, *chan*) tty revoke flag** *revoke flag* **ret** *ret*

(*maj***, *min*, *chan*) tty mpx ret** *ret*

(*maj***, *min*, *chan*) tty input c** *c input status* **ret** *ret*

(*maj***, *min*, *chan*) tty output** *output status*

(*maj***, *min*, *chan*) tty service proc** *proc* **ret** *ret*

(*maj***, *min*, *chan*) tty service set control** *control* **ret** *ret*

(*maj***, *min*, *chan*) tty service get control ret** *ret*

(*maj***, *min*, *chan*) tty service get status ret** *ret*

(*maj***, *min*, *chan*) tty service baud** *baud* **ret** *ret*

(*maj***, *min*, *chan*) tty service get baud ret** *ret*

(*maj***, *min*, *chan*) tty service set input baud** *baud* **ret** *ret*

(*maj***, *min*, *chan*) tty service get input baud ret** *ret*

(*maj***, *min*, *chan*) tty service set bpc** *bpc* **ret** *ret*

(*maj***, *min*, *chan*) tty service get bpc ret** *ret*

(*maj***, *min*, *chan*) tty service set parity** *parity* **ret** *ret*

(*maj***, *min*, *chan*) tty service get parity ret** *ret*

(*maj***, *min*, *chan*) tty service set stops** *stops* **ret** *ret*

(*maj***, *min*, *chan*) tty service get stops ret** *ret*

(*maj***, *min*, *chan*) tty service set break ret** *ret*

(*maj***, *min*, *chan*) tty service clear break ret** *ret*

(*maj***, *min*, *chan*) tty service open** *open* **ret** *ret*

(*maj***, *min*, *chan*) tty service dopace** *dopace* **ret** *ret*

(*maj***, *min*, *chan*) tty service softpace** *softpace* **ret** *ret*

(*maj***, *min*, *chan*) tty service softrchar** *softrchar* **ret** *ret*

(*maj***, *min*, *chan*) tty service softlchar** *softlchar* **ret** *ret*

(*maj***, *min*, *chan*) tty service softrstr** *softrstr* **ret** *ret*

(*maj***,** *min***,** *chan***) tty service softlstr** *softlstr* **ret** *ret*

(*maj***,** *min***,** *chan***) tty service hardrbits** *hardrbits* **ret** *ret*

(*maj***,** *min***,** *chan***) tty service hardlbits** *hardlbits* **ret** *ret*

(*maj***,** *min***,** *chan***) tty service loop enter ret** *ret*

(*maj***,** *min***,** *chan***) tty service loop exit ret** *ret*

(*maj***,** *min***,** *chan***) tty proc** *proc* **ret** *ret*

(*maj***,** *min***,** *chan***) tty slih intr** *intr slih status*

(*maj***,** *min***,** *chan***) tty offlevel intr** *intr* **ret** *ret*

(*maj***,** *min***,** *chan***) tty ttyinit disp** *disp* **ret** *ret*

(*maj***,** *min***,** *chan***) tty ttyfree ret** *ret*

(*maj***,** *min***,** *chan***) tty ttynull ret** *ret*

(*maj***,** *min***,** *chan***) tty ttcwait** *wait* **ret** *ret*

(*maj***,** *min***,** *chan***) tty ttyspgrp ret** *ret*

(*maj***,** *min***,** *chan***) tty ttypath input** *ttypath input* **ret** *ret*

(*maj***,** *min***,** *chan***) tty ttypath output** *ttypath output* **ret** *ret*

(*maj***,** *min***,** *chan***) tty ttypath service** *ttypath service* **ret** *ret*

(*maj***,** *min***,** *chan***) tty stack ctl disp** *disp* **mode** *mode* **ext** *ext* **ret** *ret*

(*maj***,** *min***,** *chan***) tty unstack ctl ctl** *ctl* **ext** *ext* **ret** *ret*

(*maj***,** *min***,** *chan***) tty getctlbytype type** *type* **ctl** *ctl* **ret** *ret*

(*maj***,** *min***,** *chan***) tty getctlbyname name** *name* **ctl** *ctl* **ret** *ret*

(*maj***,** *min***,** *chan***) tty getdispbyname name** *name* **disp** *disp* **ret** *ret*

(*maj***,** *min***,** *chan***) tty getdispbytype type** *type* **disp** *disp* **ret** *ret*

(*maj***,** *min***,** *chan***) tty dispadd ret** *ret*

(*maj***,** *min***,** *chan***) tty dispdel** *ret*

(*maj***,** *min***,** *chan***) tty tty_do_offlevel** *ret*

(*maj***,** *min***,** *chan***) tty ttyofflevel** *ret*

(*maj***,** *min***,** *chan***)** Major and minor device number, and channel number

config cmd *cmd*

cin *cin*

cmd *alloc cmd*
Possible values:

push

pop

unconfig

mode *open mode*

ext *ext*

ioctl cmd *ioctl cmd*

arg *ioctl arg*

mode *mode*

events *events*
Possible values:

in

out

pri

sync

revents *revents*

revoke flag *revoke flag*

c *c*

input status
Possible values:

good char

overrun

parity error

framing error

break interrupt

cts on

cts off

dsr on

dsr off

ri on

ri off

cd on

cd off

cblock buf

other buf

proc *proc*	Possible values: **output** **suspend** **resume** **block** **unblock** **rflush** **wflush**
output *output status*	Possible values: **done** **more output**
set control *control*	Possible values: **TSDTR** **TSRTS** **TSCTS** **TSDSR** **TSRI** **TSCD** **baud** *baud* **bpc** *bpc*
parity *parity*	Possible values: **Tnone** **odd** **mark** **even** **space**
stops *stops*	Possible values: **1** **2**
open *open*	Possible values: **local** **remote**

dopace *dopace*	Possible values: **again** **xon** **str** **dtr** **rts**
softpace *softpace*	Possible values: **remote off** **remote any** **remote on** **remote str** **local off** **local on** **local str** **softrchar** *softrchar* **softlchar** *softlchar* **softrstr** *softrstr* **softlstr** *softlstr* **hardrbits** *hardrbits* **hardlbits** *hardlbits*
slih status	Possible values: **serviced** **no intr serviced** **intr** *intr* **disp** *disp* **ttcwait** *wait* **ttypath input** *ttypath input* **ttypath output** *ttypath output* **ttypath service** *ttypath service* **ctl** *ctl* **type** *type* **name** *name* **ret** *ret*

Trace Hook IDs: 402

The following trace hook IDs are stored in the **/usr/include/sys/trchkid.h** file.

402 : HKWD TTY PTY

Recorded Data

Event:

(*maj***,** *min***,** *chan***) pty config cmd** *cmd* **ret** *ret*

(*maj***,** *min***,** *chan***) pty alloc cin** *cin* **cmd** *alloc cmd* **ret** *ret*

(*maj***,** *min***,** *chan***) pty open mode** *open mode* **ext** *ext* **ret** *ret*

(*maj***,** *min***,** *chan***) pty close ret** *ret*

(*maj***,** *min***,** *chan***) pty read ret** *ret*

(*maj***,** *min***,** *chan***) pty write ret** *ret*

(*maj***,** *min***,** *chan***) pty ioctl cmd** *ioctl cmd* **arg** *ioctl arg* **mode** *mode* **ret** *ret*

(*maj***,** *min***,** *chan***) pty select events** *events* **revents** *revents* **ret** *ret*

(*maj***,** *min***,** *chan***) pty revoke flag** *revoke flag* **ret** *ret*

(*maj***,** *min***,** *chan***) pty mpx ret** *ret*

(*maj***,** *min***,** *chan***) pty input c** *c input status* **ret** *ret*

(*maj***,** *min***,** *chan***) pty output** *output status*

(*maj***,** *min***,** *chan***) pty service proc** *proc* **ret** *ret*

(*maj***,** *min***,** *chan***) pty service set control** *control* **ret** *ret*

(*maj***,** *min***,** *chan***) pty service get control ret** *ret*

(*maj***,** *min***,** *chan***) pty service get status ret** *ret*

(*maj***,** *min***,** *chan***) pty service baud** *baud* **ret** *ret*

(*maj***,** *min***,** *chan***) pty service get baud ret** *ret*

(*maj***,** *min***,** *chan***) pty service set input baud** *baud* **ret** *ret*

(*maj***,** *min***,** *chan***) pty service get input baud ret** *ret*

(*maj***,** *min***,** *chan***) pty service set bpc** *bpc* **ret** *ret*

(*maj***,** *min***,** *chan***) pty service get bpc ret** *ret*

(*maj***,** *min***,** *chan***) pty service set parity** *parity* **ret** *ret*

(*maj***, *min*, *chan*) pty service get parity ret** *ret*

(*maj***, *min*, *chan*) pty service set stops** *stops* **ret** *ret*

(*maj***, *min*, *chan*) pty service get stops ret** *ret*

(*maj***, *min*, *chan*) pty service set break ret** *ret*

(*maj***, *min*, *chan*) pty service clear break ret** *ret*

(*maj***, *min*, *chan*) pty service open** *open* **ret** *ret*

(*maj***, *min*, *chan*) pty service dopace** *dopace* **ret** *ret*

(*maj***, *min*, *chan*) pty service softpace** *softpace* **ret** *ret*

(*maj***, *min*, *chan*) pty service softrchar** *softrchar* **ret** *ret*

(*maj***, *min*, *chan*) pty service softlchar** *softlchar* **ret** *ret*

(*maj***, *min*, *chan*) pty service softrstr** *softrstr* **ret** *ret*

(*maj***, *min*, *chan*) pty service softlstr** *softlstr* **ret** *ret*

(*maj***, *min*, *chan*) pty service hardrbits** *hardrbits* **ret** *ret*

(*maj***, *min*, *chan*) pty service hardlbits** *hardlbits* **ret** *ret*

(*maj***, *min*, *chan*) pty service loop enter ret** *ret*

(*maj***, *min*, *chan*) pty service loop exit ret** *ret*

(*maj***, *min*, *chan*) pty proc** *proc* **ret** *ret*

(*maj***, *min*, *chan*) pty slih intr** *intr slih status*

(*maj***, *min*, *chan*) pty offlevel intr** *intr* **ret** *ret*

(*maj***, *min*, *chan*) pty ptycreate ret** *ret*

(*maj***, *min*, *chan*) pty ptcwakeup flag** *flag* **ret** *ret*

(*maj***, *min*, *chan*)** Major and minor device number, and channel number.

config cmd *cmd*

cin *cin*

cmd *alloc cmd*
Possible values:

push

pop

unconfig

mode *open mode*

ext *ext*

ioctl cmd *ioctl cmd*

arg *ioctl arg*

mode *mode*

events *events*
Possible values:

in

out

pri

sync

revents *revents*

revoke flag *revoke flag*

c *c*

input status
Possible values:

good char

overrun

parity error

framing error

break interrupt

cts on

cts off

dsr on

dsr off

ri on

ri off

cd on

cd off

cblock buf

other buf

proc *proc*	Possible values:
	output
	suspend
	resume
	block
	unblock
	rflush
	wflush
output *output status*	Possible values:
	done
	more output
set control *control*	Possible values:
	TSDTR
	TSRTS
	TSCTS
	TSDSR
	TSRI
	TSCD
	baud *baud*
	bpc *bpc*
parity *parity*	Possible values:
	none
	odd
	mark
	even
	space
stops *stops*	Possible values:
	1
	2
open *open*	Possible values:
	local
	remote

dopace *dopace*	Possible values: **again** **xon** **str** **dtr** **rts**
softpace *softpace*	Possible values: **remote off** **remote any** **remote on** **remote str** **local off** **local on** **local str** **softrchar** *softrchar* **softlchar** *softlchar* **softrstr** *softrstr* **softlstr** *softlstr* **hardrbits** *hardrbits* **hardlbits** *hardlbits* **intr** *intr* **flag** *flag* *slih status* **serviced** **no intr serviced** **ret** *ret*

Trace Hook IDs: 403

The following trace hook IDs are stored in the **/usr/include/sys/trchkid.h** file.

403 : HKWD TTY RS

Recorded Data

Event:

(*maj***,** *min***,** *chan***) rs config cmd** *cmd* **ret** *ret*

(*maj***,** *min***,** *chan***) rs alloc cin** *cin* **cmd** *alloc cmd* **ret** *ret*

(*maj***,** *min***,** *chan***) rs open mode** *open mode* **ext** *ext* **ret** *ret*

(*maj***,** *min***,** *chan***) rs close ret** *ret*

(*maj***,** *min***,** *chan***) rs read ret** *ret*

(*maj***,** *min***,** *chan***) rs write ret** *ret*

(*maj***,** *min***,** *chan***) rs ioctl cmd** *ioctl cmd* **arg** *ioctl arg* **mode** *mode* **ret** *ret*

(*maj***,** *min***,** *chan***) rs select events** *events* **revents** *revents* **ret** *ret*

(*maj***,** *min***,** *chan***) rs revoke flag** *revoke flag* **ret** *ret*

(*maj***,** *min***,** *chan***) rs mpx ret** *ret*

(*maj***,** *min***,** *chan***) rs input c** *c input status* **ret** *ret*

(*maj***,** *min***,** *chan***) rs output** *output status*

(*maj***,** *min***,** *chan***) rs service proc** *proc* **ret** *ret*

(*maj***,** *min***,** *chan***) rs service set control** *control* **ret** *ret*

(*maj***,** *min***,** *chan***) rs service get control ret** *ret*

(*maj***,** *min***,** *chan***) rs service get status ret** *ret*

(*maj***,** *min***,** *chan***) rs service baud** *baud* **ret** *ret*

(*maj***,** *min***,** *chan***) rs service get baud ret** *ret*

(*maj***,** *min***,** *chan***) rs service set input baud** *baud* **ret** *ret*

(*maj***,** *min***,** *chan***) rs service get input baud ret** *ret*

(*maj***,** *min***,** *chan***) rs service set bpc** *bpc* **ret** *ret*

(*maj***,** *min***,** *chan***) rs service get bpc ret** *ret*

(*maj***,** *min***,** *chan***) rs service set parity** *parity* **ret** *ret*

(*maj***,** *min***,** *chan***) rs service get parity ret** *ret*

(*maj***,** *min***,** *chan***) rs service set stops** *stops* **ret** *ret*

(*maj***,** *min***,** *chan***) rs service get stops ret** *ret*

(*maj***,** *min***,** *chan***) rs service set break ret** *ret*

(*maj***,** *min***,** *chan***) rs service clear break ret** *ret*

(*maj***,** *min***,** *chan***) rs service open** *open* **ret** *ret*

(*maj***,** *min***,** *chan***) rs service dopace** *dopace* **ret** *ret*

(*maj***,** *min***,** *chan***) rs service softpace** *softpace* **ret** *ret*

(*maj***,** *min***,** *chan***) rs service softrchar** *softrchar* **ret** *ret*

(*maj***,** *min***,** *chan***) rs service softlchar** *softlchar* **ret** *ret*

(*maj***,** *min***,** *chan***) rs service softrstr** *softrstr* **ret** *ret*

(*maj***,** *min***,** *chan***) rs service softlstr** *softlstr* **ret** *ret*

(*maj***,** *min***,** *chan***) rs service hardrbits** *hardrbits* **ret** *ret*

(*maj***,** *min***,** *chan***) rs service hardlbits** *hardlbits* **ret** *ret*

(*maj***,** *min***,** *chan***) rs service loop enter ret** *ret*

(*maj***,** *min***,** *chan***) rs service loop exit ret** *ret*

(*maj***,** *min***,** *chan***) rs proc** *proc* **ret** *ret*

(*maj***,** *min***,** *chan***) rs slih intr** *intr slih status*

(*maj***,** *min***,** *chan***) rs offlevel intr** *intr* **ret** *ret*

(*maj***,** *min***,** *chan***) rs add type** *type* **ret** *ret*

(*maj***,** *min***,** *chan***) rs delete type** *type* **ret** *ret*

(*maj***,** *min***,** *chan***) rs nslih intr** *intr* **ret** *ret*

(*maj***,** *min***,** *chan***) rs 8slih intr** *intr* **ret** *ret*

(*maj***,** *min***,** *chan***) rs RT8slih intr** *intr* **ret** *ret*

(*maj***,** *min***,** *chan***) rs RT4slih intr** *intr* **ret** *ret*

(*maj***,** *min***,** *chan***) rs RT4detect id_ptr** *id_ptr* **ret** *ret*

(*maj***,** *min***,** *chan***)** Major and minor device number, and channel number.

config cmd *cmd*

cin *cin*

cmd *alloc cmd*
Possible values:

push

pop

unconfig

mode *open mode*

ext *ext*

ioctl cmd *ioctl cmd*

arg *ioctl arg*

mode *mode*

events *events*
Possible values:

in

out

pri

sync

revents *revents*

revoke flag *revoke flag*

c *c*

input status
Possible values:

good char

overrun

parity error

framing error

break interrupt

cts on

cts off

dsr on

dsr off

ri on

ri off

cd on

cd off

cblock buf

other buf

proc *proc*

Possible values:

output

suspend

resume

block

unblock

rflush

wflush

output *output status*

Possible values:

done

more output

set control *control*

Possible values:

TSDTR

TSRTS

TSCTS

TSDSR

TSRI

TSCD

baud *baud*

bpc *bpc*

parity *parity*

Possible values:

none

odd

mark

even

space

stops *stops*

Possible values:

1

2

open *open*

Possible values:

local

remote

dopace *dopace*	Possible values: **again** **xon** **str** **dtr** **rts**
softpace *softpace*	Possible values: **remote off** **remote any** **remote on** **remote str** **local off** **local on** **local str** **softrchar** *softrchar* **softlchar** *softlchar* **softrstr** *softrstr* **softlstr** *softlstr* **hardrbits** *hardrbits* **hardlbits** *hardlbits* **intr** *intr* **type** *type*
slih status	Possible values: **servicedno intr servicedid_ptr** *id_prt* **ret** *ret*

Trace Hook IDs: 404

The following trace hook IDs are stored in the **/usr/include/sys/trchkid.h** file.

404 : HKWD TTY LION

Recorded Data

Event:

(*maj***,** *min***,** *chan***) lion config cmd** *cmd* **ret** *ret*

(*maj***,** *min***,** *chan***) lion alloc cin** *cin* **cmd** *alloc cmd* **ret** *ret*

(*maj*, *min*, *chan*) **lion open mode** *open mode* **ext** *ext* **ret** *ret*

(*maj*, *min*, *chan*) **lion close ret** *ret*

(*maj*, *min*, *chan*) **lion read ret** *ret*

(*maj*, *min*, *chan*) **lion write ret** *ret*

(*maj*, *min*, *chan*) **lion ioctl cmd** *ioctl cmd* **arg** *ioctl arg* **mode** *mode* **ret** *ret*

(*maj*, *min*, *chan*) **lion select events** *events* **revents** *revents* **ret** *ret*

(*maj*, *min*, *chan*) **lion revoke flag** *revoke flag* **ret** *ret*

(*maj*, *min*, *chan*) **lion mpx ret** *ret*

(*maj*, *min*, *chan*) **lion input c** *c input status* **ret** *ret*

(*maj*, *min*, *chan*) **lion output** *output status*

(*maj*, *min*, *chan*) **lion service proc** *proc* **ret** *ret*

(*maj*, *min*, *chan*) **lion service set control** *control* **ret** *ret*

(*maj*, *min*, *chan*) **lion service get control ret** *ret*

(*maj*, *min*, *chan*) **lion service get status ret** *ret*

(*maj*, *min*, *chan*) **lion service baud** *baud* **ret** *ret*

(*maj*, *min*, *chan*) **lion service get baud ret** *ret*

(*maj*, *min*, *chan*) **lion service set input baud** *baud* **ret** *ret*

(*maj*, *min*, *chan*) **lion service get input baud ret** *ret*

(*maj*, *min*, *chan*) **lion service set bpc** *bpc* **ret** *ret*

(*maj*, *min*, *chan*) **lion service get bpc ret** *ret*

(*maj*, *min*, *chan*) **lion service set parity** *parity* **ret** *ret*

(*maj*, *min*, *chan*) **lion service get parity ret** *ret*

(*maj*, *min*, *chan*) **lion service set stops** *stops* **ret** *ret*

(*maj*, *min*, *chan*) **lion service get stops ret** *ret*

(*maj*, *min*, *chan*) **lion service set break ret** *ret*

(*maj*, *min*, *chan*) **lion service clear break ret** *ret*

(*maj*, *min*, *chan*) **lion service open** *open* **ret** *ret*

(*maj*, *min*, *chan*) **lion service dopace** *dopace* **ret** *ret*

(*maj***,** *min***,** *chan***) lion service softpace** *softpace* **ret** *ret*

(*maj***,** *min***,** *chan***) lion service softrchar** *softrchar* **ret** *ret*

(*maj***,** *min***,** *chan***) lion service softlchar** *softlchar* **ret** *ret*

(*maj***,** *min***,** *chan***) lion service softrstr** *softrstr* **ret** *ret*

(*maj***,** *min***,** *chan***) lion service softlstr** *softlstr* **ret** *ret*

(*maj***,** *min***,** *chan***) lion service hardrbits** *hardrbits* **ret** *ret*

(*maj***,** *min***,** *chan***) lion service hardlbits** *hardlbits* **ret** *ret*

(*maj***,** *min***,** *chan***) lion service loop enter ret** *ret*

(*maj***,** *min***,** *chan***) lion service loop exit ret** *ret*

(*maj***,** *min***,** *chan***) lion proc** *proc* **ret** *ret*

(*maj***,** *min***,** *chan***) lion slih intr** *intr slih status*

(*maj***,** *min***,** *chan***) lion offlevel intr** *intr* **ret** *ret*

(*maj***,** *min***,** *chan***) lion add ret** *ret*

(*maj***,** *min***,** *chan***) lion add del** *ret*

(*maj***,** *min***,** *chan***)** Major and minor device number, and channel number.

config cmd *cmd*

cin *cin*

cmd *alloc cmd* Possible values:

push

pop

unconfig

mode *open mode*

ext *ext*

ioctl cmd *ioctl cmd*

arg *ioctl arg*

mode *mode*

events *events*

Possible values:

in

out

pri

sync

revents *revents*

revoke flag *revoke flag*

c *c*

input status

Possible values:

good char

overrun

parity error

framing error

break interrupt

cts on

cts off

dsr on

dsr off

ri on

ri off

cd on

cd off

cblock buf

other buf

proc *proc*

Possible values:

output

suspend

resume

block

unblock

rflush

wflush

output *output status*

Possible values:

done

more output

set control *control*	Possible values:
	TSDTR
	TSRTS
	TSCTS
	TSDSR
	TSRI
	TSCD
	baud *baud*
	bpc *bpc*
parity *parity*	Possible values:
	none
	odd
	mark
	even
	space
stops *stops*	Possible values:
	1
	2
open *open*	Possible values:
	local
	remote
dopace *dopace*	Possible values:
	again
	xon
	str
	dtr
	rts

softpace *softpace*	Possible values: **remote off** **remote any** **remote on** **remote str** **local off** **local on** **local str** **softrchar** *softrchar* **softlchar** *softlchar* **softrstr** *softrstr* **softlstr** *softlstr* **hardrbits** *hardrbits* **hardlbits** *hardlbits* **intr** *intr*
slih status	Possible values: **serviced** **no intr serviced** **ret** *ret*

Trace Hook IDs: 405

The following trace hook IDs are stored in the **/usr/include/sys/trchkid.h** file.

405 : HKWD TTY HFT

Recorded Data

Event:

(*maj***,** *min***,** *chan***) hft config cmd** *cmd* **ret** *ret*

(*maj***,** *min***,** *chan***) hft alloc cin** *cin* **cmd** *alloc cmd* **ret** *ret*

(*maj***,** *min***,** *chan***) hft open mode** *open mode* **ext** *ext* **ret** *ret*

(*maj***,** *min***,** *chan***) hft close ret** *ret*

(*maj***,** *min***,** *chan***) hft read ret** *ret*

(*maj***,** *min***,** *chan***) hft write ret** *ret*

(*maj***,** *min***,** *chan***) hft ioctl cmd** *ioctl cmd* **arg** *ioctl arg* **mode** *mode* **ret** *ret*

(*maj***, *min*, *chan*) hft select events** *events* **revents** *revents* **ret** *ret*

(*maj***, *min*, *chan*) hft revoke flag** *revoke flag* **ret** *ret*

(*maj***, *min*, *chan*) hft mpx ret** *ret*

(*maj***, *min*, *chan*) hft input c** *c input status* **ret** *ret*

(*maj***, *min*, *chan*) hft output** *output status*

(*maj***, *min*, *chan*) hft service proc** *proc* **ret** *ret*

(*maj***, *min*, *chan*) hft service set control** *control* **ret** *ret*

(*maj***, *min*, *chan*) hft service get control ret** *ret*

(*maj***, *min*, *chan*) hft service get status ret** *ret*

(*maj***, *min*, *chan*) hft service baud** *baud* **ret** *ret*

(*maj***, *min*, *chan*) hft service get baud ret** *ret*

(*maj***, *min*, *chan*) hft service set input baud** *baud* **ret** *ret*

(*maj***, *min*, *chan*) hft service get input baud ret** *ret*

(*maj***, *min*, *chan*) hft service set bpc** *bpc* **ret** *ret*

(*maj***, *min*, *chan*) hft service get bpc ret** *ret*

(*maj***, *min*, *chan*) hft service set parity** *parity* **ret** *ret*

(*maj***, *min*, *chan*) hft service get parity ret** *ret*

(*maj***, *min*, *chan*) hft service set stops** *stops* **ret** *ret*

(*maj***, *min*, *chan*) hft service get stops ret** *ret*

(*maj***, *min*, *chan*) hft service set break ret** *ret*

(*maj***, *min*, *chan*) hft service clear break ret** *ret*

(*maj***, *min*, *chan*) hft service open** *open* **ret** *ret*

(*maj***, *min*, *chan*) hft service dopace** *dopace* **ret** *ret*

(*maj***, *min*, *chan*) hft service softpace** *softpace* **ret** *ret*

(*maj***, *min*, *chan*) hft service softrchar** *softrchar* **ret** *ret*

(*maj***, *min*, *chan*) hft service softlchar** *softlchar* **ret** *ret*

(*maj***, *min*, *chan*) hft service softrstr** *softrstr* **ret** *ret*

(*maj***, *min*, *chan*) hft service softlstr** *softlstr* **ret** *ret*

(*maj***,** *min***,** *chan***) hft service hardrbits** *hardrbits* **ret** *ret*

(*maj***,** *min***,** *chan***) hft service hardlbits** *hardlbits* **ret** *ret*

(*maj***,** *min***,** *chan***) hft service loop enter ret** *ret*

(*maj***,** *min***,** *chan***) hft service loop exit ret** *ret*

(*maj***,** *min***,** *chan***) hft proc** *proc* **ret** *ret*

(*maj***,** *min***,** *chan***) hft slih intr** *intr slih status*

(*maj***,** *min***,** *chan***) hft offlevel intr** *intr* **ret** *ret*

(*maj***,** *min***,** *chan***)** Major and minor device number, and channel number.

config cmd *cmd*

cin *cin*

cmd *alloc cmd* Possible values:

push

pop

unconfig

mode *open mode*

ext *ext*

ioctl cmd *ioctl cmd*

arg *ioctl arg*

mode *mode*

events *events* Possible values:

in

out

pri

sync

revents *revents*

revoke flag *revoke flag*

c *c*

input status	Possible values: **good char** **overrun** **parity error** **framing error** **break interrupt** **cts on** **cts off** **dsr on** **dsr off** **ri on** **ri off** **cd on** **cd offc** **block buf** **other buf**
proc *proc*	Possible values: **output** **suspend** **resume** **block** **unblock** **rflush** **wflush**
output *output status*	Possible values: **done** **more output**

set control *control*
Possible values:

TSDTR

TSRTS

TSCTS

TSDSR

TSRI

TSCD

baud *baud*

bpc *bpc*

parity *parity*
Possible values:

none

odd

mark

even

space

stops *stops*
Possible values:

1

2

open *open*
Possible values:

local

remote

dopace *dopace*
Possible values:

again

xon

str

dtr

rts

softpace *softpace*	Possible values:
	remote off
	remote any
	remote on
	remote str
	local off
	local on
	local str
	softrchar *softrchar*
	softlchar *softlchar*
	softrstr *softrstr*
	softlstr *softlstr*
	hardrbits *hardrbits*
	hardlbits *hardlbits*
	intr *intr*
slih status	Possible values:
	serviced
	no intr serviced
	ret *ret*

Trace Hook IDs: 406

The following trace hook IDs are stored in the **/usr/include/sys/trchkid.h** file.

406 : HKWD TTY RTS

Recorded Data

Event:

(*maj***,** *min***,** *chan***) rts config cmd** *cmd* **ret** *ret*

(*maj***,** *min***,** *chan***) rts alloc cin** *cin* **cmd** *alloc cmd* **ret** *ret*

(*maj***,** *min***,** *chan***) rts open mode** *open mode* **ext** *ext* **ret** *ret*

(*maj***,** *min***,** *chan***) rts close ret** *ret*

(*maj***,** *min***,** *chan***) rts read ret** *ret*

(*maj***,** *min***,** *chan***) rts write ret** *ret*

(*maj***,** *min***,** *chan***) rts ioctl cmd** *ioctl cmd* **arg** *ioctl arg* **mode** *mode* **ret** *ret*

(*maj*, *min*, *chan*) **rts select events** *events* **revents** *revents* **ret** *ret*

(*maj*, *min*, *chan*) **rts revoke flag** *revoke flag* **ret** *ret*

(*maj*, *min*, *chan*) **rts mpx ret** *ret*

(*maj*, *min*, *chan*) **rts input c** *c input status* **ret** *ret*

(*maj*, *min*, *chan*) **rts output** *output status*

(*maj*, *min*, *chan*) **rts service proc** *proc* **ret** *ret*

(*maj*, *min*, *chan*) **rts service set control** *control* **ret** *ret*

(*maj*, *min*, *chan*) **rts service get control ret** *ret*

(*maj*, *min*, *chan*) **rts service get status ret** *ret*

(*maj*, *min*, *chan*) **rts service baud** *baud* **ret** *ret*

(*maj*, *min*, *chan*) **rts service get baud ret** *ret*

(*maj*, *min*, *chan*) **rts service set input baud** *baud* **ret** *ret*

(*maj*, *min*, *chan*) **rts service get input baud ret** *ret*

(*maj*, *min*, *chan*) **rts service set bpc** *bpc* **ret** *ret*

(*maj*, *min*, *chan*) **rts service get bpc ret** *ret*

(*maj*, *min*, *chan*) **rts service set parity** *parity* **ret** *ret*

(*maj*, *min*, *chan*) **rts service get parity ret** *ret*

(*maj*, *min*, *chan*) **rts service set stops** *stops* **ret** *ret*

(*maj*, *min*, *chan*) **rts service get stops ret** *ret*

(*maj*, *min*, *chan*) **rts service set break ret** *ret*

(*maj*, *min*, *chan*) **rts service clear break ret** *ret*

(*maj*, *min*, *chan*) **rts service open** *open* **ret** *ret*

(*maj*, *min*, *chan*) **rts service dopace** *dopace* **ret** *ret*

(*maj*, *min*, *chan*) **rts service softpace** *softpace* **ret** *ret*

(*maj*, *min*, *chan*) **rts service softrchar** *softrchar* **ret** *ret*

(*maj*, *min*, *chan*) **rts service softlchar** *softlchar* **ret** *ret*

(*maj*, *min*, *chan*) **rts service softrstr** *softrstr* **ret** *ret*

(*maj*, *min*, *chan*) **rts service softlstr** *softlstr* **ret** *ret*

(*maj***,** *min***,** *chan***) rts service hardrbits** *hardrbits* **ret** *ret*

(*maj***,** *min***,** *chan***) rts service hardlbits** *hardlbits* **ret** *ret*

(*maj***,** *min***,** *chan***) rts service loop enter ret** *ret*

(*maj***,** *min***,** *chan***) rts service loop exit ret** *ret*

(*maj***,** *min***,** *chan***) rts proc** *proc* **ret** *ret*

(*maj***,** *min***,** *chan***) rts slih intr** *intr slih status*

(*maj***,** *min***,** *chan***) rts offlevel intr** *intr* **ret** *ret*

(*maj***,** *min***,** *chan***)** Major and minor device number, and channel number.

config cmd *cmd*

cin *cin*

cmd *alloc cmd* Possible values:

push

pop

unconfig

mode *open mode*

ext *ext*

ioctl cmd *ioctl cmd*

arg *ioctl arg*

mode *mode*

events *events* Possible values:

in

out

pri

sync

revents *revents*

revoke flag *revoke flag*

c *cinput status*	Possible values: **good char** **overrun** **parity error** **framing error** **break interrupt** **cts on** **cts off** **dsr on** **dsr off** **ri on** **ri off** **cd on** **cd off** **cblock buf** **other buf**
proc *proc*	Possible values: **output** **suspend** **resume** **block** **unblock** **rflush** **wflush**
output *output status*	Possible values: **done** **more output**

set control *control*	Possible values:
	TSDTR
	TSRTS
	TSCTS
	TSDSR
	TSRI
	TSCD
	baud *baud*
	bpc *bpc*
parity *parity*	Possible values:
	none
	odd
	mark
	even
	space
stops *stops*	Possible values:
	1
	2
open *open*	Possible values:
	local
	remote
dopace *dopace*	Possible values:
	again
	xon
	str
	dtr
	rts

softpace *softpace*	Possible values:
	remote off
	remote any
	remote on
	remote str
	local off
	local on
	local str
	softrchar *softrchar*
	softlchar *softlchar*
	softrstr *softrstr*
	softlstr *softlstr*
	hardrbits *hardrbits*
	hardlbits *hardlbits*
	intr *intr*
slih status	Possible values:
	serviced
	no intr serviced
	ret *ret*

Trace Hook IDs: 407

The following trace hook IDs are stored in the **/usr/include/sys/trchkid.h** file.

407 : HKWD TTY XON

Recorded Data

Event:

(*maj***,** *min***,** *chan***) xon config cmd** *cmd* **ret** *ret*

(*maj***,** *min***,** *chan***) xon alloc cin** *cin* **cmd** *alloc cmd* **ret** *ret*

(*maj***,** *min***,** *chan***) xon open mode** *open mode* **ext** *ext* **ret** *ret*

(*maj***,** *min***,** *chan***) xon close ret** *ret*

(*maj***,** *min***,** *chan***) xon read ret** *ret*

(*maj***,** *min***,** *chan***) xon write ret** *ret*

(*maj***,** *min***,** *chan***) xon ioctl cmd** *ioctl cmd* **arg** *ioctl arg* **mode** *mode* **ret** *ret*

(*maj***, *min*, *chan*) xon select events** *events* **revents** *revents* **ret** *ret*

(*maj***, *min*, *chan*) xon revoke flag** *revoke flag* **ret** *ret*

(*maj***, *min*, *chan*) xon mpx ret** *ret*

(*maj***, *min*, *chan*) xon input c** *c input status* **ret** *ret*

(*maj***, *min*, *chan*) xon output** *output status*

(*maj***, *min*, *chan*) xon service proc** *proc* **ret** *ret*

(*maj***, *min*, *chan*) xon service set control** *control* **ret** *ret*

(*maj***, *min*, *chan*) xon service get control ret** *ret*

(*maj***, *min*, *chan*) xon service get status ret** *ret*

(*maj***, *min*, *chan*) xon service baud** *baud* **ret** *ret*

(*maj***, *min*, *chan*) xon service get baud ret** *ret*

(*maj***, *min*, *chan*) xon service set input baud** *baud* **ret** *ret*

(*maj***, *min*, *chan*) xon service get input baud ret** *ret*

(*maj***, *min*, *chan*) xon service set bpc** *bpc* **ret** *ret*

(*maj***, *min*, *chan*) xon service get bpc ret** *ret*

(*maj***, *min*, *chan*) xon service set parity** *parity* **ret** *ret*

(*maj***, *min*, *chan*) xon service get parity ret** *ret*

(*maj***, *min*, *chan*) xon service set stops** *stops* **ret** *ret*

(*maj***, *min*, *chan*) xon service get stops ret** *ret*

(*maj***, *min*, *chan*) xon service set break ret** *ret*

(*maj***, *min*, *chan*) xon service clear break ret** *ret*

(*maj***, *min*, *chan*) xon service open** *open* **ret** *ret*

(*maj***, *min*, *chan*) xon service dopace** *dopace* **ret** *ret*

(*maj***, *min*, *chan*) xon service softpace** *softpace* **ret** *ret*

(*maj***, *min*, *chan*) xon service softrchar** *softrchar* **ret** *ret*

(*maj***, *min*, *chan*) xon service softlchar** *softlchar* **ret** *ret*

(*maj***, *min*, *chan*) xon service softrstr** *softrstr* **ret** *ret*

(*maj***, *min*, *chan*) xon service softlstr** *softlstr* **ret** *ret*

(*maj***,** *min***,** *chan***) xon service hardrbits** *hardrbits* **ret** *ret*

(*maj***,** *min***,** *chan***) xon service hardlbits** *hardlbits* **ret** *ret*

(*maj***,** *min***,** *chan***) xon service loop enter ret** *ret*

(*maj***,** *min***,** *chan***) xon service loop exit ret** *ret*

(*maj***,** *min***,** *chan***) xon proc** *proc* **ret** *ret*

(*maj***,** *min***,** *chan***) xon slih intr** *intr slih status*

(*maj***,** *min***,** *chan***) xon offlevel intr** *intr* **ret** *ret*

(*maj***,** *min***,** *chan***)** Major and minor device number, and channel number.

config cmd *cmd*

cin *cin*

cmd *alloc cmd* Possible values:

push

pop

unconfig

mode *open mode*

ext *ext*

ioctl cmd *ioctl cmd*

arg *ioctl arg*

mode *mode*

events *events* Possible values:

in

out

pri

sync

revents *revents*

revoke flag *revoke flag*

c *c*

input status	Possible values: **good char** **overrun** **parity error** **framing error** **break interrupt** **cts on** **cts off** **dsr on** **dsr off** **ri on** **ri off** **cd on** **cd off** **cblock buf** **other buf**
proc *proc*	Possible values: **output** **suspend** **resume** **block** **unblock** **rflush** **wflush**
output *output status*	Possible values: **done** **more output**

set control *control*
Possible values:

TSDTR

TSRTS

TSCTS

TSDSR

TSRI

TSCD

baud *baud*

bpc *bpc*

parity *parity*
Possible values:

none

odd

mark

even

space

stops *stops*
Possible values:

1

2

open *open*
Possible values:

local

remote

dopace *dopace*
Possible values:

again

xon

str

dtr

rts

softpace *softpace*	Possible values: **remote off** **remote any** **remote on** **remote str** **local off** **local on** **local str** **softrchar** *softrchar* **softlchar** *softlchar* **softrstr** *softrstr* **softlstr** *softlstr* **ardrbits** *hardrbits* **hardlbits** *hardlbits* **intr** *intr*
slih status	Possible values: **serviced** **no intr serviced**

Trace Hook IDs: 408

The following trace hook IDs are stored in the **/usr/include/sys/trchkid.h** file.

408 : HKWD TTY DTR

Recorded Data

Event:

(*maj***,** *min***,** *chan***) dtr config cmd** *cmd* **ret** *ret*

(*maj***,** *min***,** *chan***) dtr alloc cin** *cin* **cmd** *alloc cmd* **ret** *ret*

(*maj***,** *min***,** *chan***) dtr open mode** *open mode* **ext** *ext* **ret** *ret*

(*maj***,** *min***,** *chan***) dtr close ret** *ret*

(*maj***,** *min***,** *chan***) dtr read ret** *ret*

(*maj***,** *min***,** *chan***) dtr write ret** *ret*

(*maj***,** *min***,** *chan***) dtr ioctl cmd** *ioctl cmd* **arg** *ioctl arg* **mode** *mode* **ret** *ret*

(*maj***,** *min***,** *chan***) dtr select events** *events* **revents** *revents* **ret** *ret*

(*maj***,** *min***,** *chan***) dtr revoke flag** *revoke flag* **ret** *ret*

(*maj*, *min*, *chan*) **dtr mpx ret** *ret*

(*maj*, *min*, *chan*) **dtr input c** *c input status* **ret** *ret*

(*maj*, *min*, *chan*) **dtr output** *output status*

(*maj*, *min*, *chan*) **dtr service proc** *proc* **ret** *ret*

(*maj*, *min*, *chan*) **dtr service set control** *control* **ret** *ret*

(*maj*, *min*, *chan*) **dtr service get control ret** *ret*

(*maj*, *min*, *chan*) **dtr service get status ret** *ret*

(*maj*, *min*, *chan*) **dtr service baud** *baud* **ret** *ret*

(*maj*, *min*, *chan*) **dtr service get baud ret** *ret*

(*maj*, *min*, *chan*) **dtr service set input baud** *baud* **ret** *ret*

(*maj*, *min*, *chan*) **dtr service get input baud ret** *ret*

(*maj*, *min*, *chan*) **dtr service set bpc** *bpc* **ret** *ret*

(*maj*, *min*, *chan*) **dtr service get bpc ret** *ret*

(*maj*, *min*, *chan*) **dtr service set parity** *parity* **ret** *ret*

(*maj*, *min*, *chan*) **dtr service get parity ret** *ret*

(*maj*, *min*, *chan*) **dtr service set stops** *stops* **ret** *ret*

(*maj*, *min*, *chan*) **dtr service get stops ret** *ret*

(*maj*, *min*, *chan*) **dtr service set break ret** *ret*

(*maj*, *min*, *chan*) **dtr service clear break ret** *ret*

(*maj*, *min*, *chan*) **dtr service open** *open* **ret** *ret*

(*maj*, *min*, *chan*) **dtr service dopace** *dopace* **ret** *ret*

(*maj*, *min*, *chan*) **dtr service softpace** *softpace* **ret** *ret*

(*maj*, *min*, *chan*) **dtr service softrchar** *softrchar* **ret** *ret*

(*maj*, *min*, *chan*) **dtr service softlchar** *softlchar* **ret** *ret*

(*maj*, *min*, *chan*) **dtr service softrstr** *softrstr* **ret** *ret*

(*maj*, *min*, *chan*) **dtr service softlstr** *softlstr* **ret** *ret*

(*maj*, *min*, *chan*) **dtr service hardrbits** *hardrbits* **ret** *ret*

(*maj*, *min*, *chan*) **dtr service hardlbits** *hardlbits* **ret** *ret*

(*maj***,** *min***,** *chan***) dtr service loop enter ret** *ret*

(*maj***,** *min***,** *chan***) dtr service loop exit ret** *ret*

(*maj***,** *min***,** *chan***) dtr proc** *proc* **ret** *ret*

(*maj***,** *min***,** *chan***) dtr slih intr** *intr slih status*

(*maj***,** *min***,** *chan***) dtr offlevel intr** *intr* **ret** *ret*

(*maj***,** *min***,** *chan***)** Major and minor device number, and channel number.

config cmd *cmd*

cin *cin*

cmd *alloc cmd* Possible values:

push

pop

unconfig

mode *open mode*

ext *ext*

ioctl cmd *ioctl cmd*

arg *ioctl arg*

mode *mode*

events *events* Possible values:

in

out

pri

sync

revents *revents*

revoke flag *revoke flag*

c *c*

input status Possible values:

good char

overrun

parity error

framing error

break interrupt

cts on

cts off

dsr on

dsr off

ri on

ri off

cd on

cd off

cblock buf

other bufproc *proc* Possible values:

output

suspend

resume

block

unblock

rflush

wflush

output *output status* Possible values:

done

more output

set control *control* Possible values:

TSDTR

TSRTS

TSCTS

TSDSR

TSRI

TSCD

baud *baud*

bpc *bpc*

parity *parity*
Possible values:

none

odd

mark

even

space

stops *stops*
Possible values:

1

2

open *open*
Possible values:

local

remote

dopace *dopace*
Possible values:

again

xon

str

dtr

rts

softpace *softpace*
Possible values:

remote off

remote any

remote on

remote str

local off

local on

local str

softrchar *softrchar*

softlchar *softlchar*

softrstr *softrstr*

softlstr *softlstr*

hardrbits *hardrbits*

hardlbits *hardlbits*

intr *intr*

slih status	Possible values: **serviced** **no intr serviced** **ret** *ret*

Trace Hook IDs: 409

The following trace hook IDs are stored in the **/usr/include/sys/trchkid.h** file.

409 : HKWD TTY DTRO

Recorded Data

Event:

(*maj***,** *min***,** *chan***) dtr open config cmd** *cmd* **ret** *ret*

(*maj***,** *min***,** *chan***) dtr open alloc cin** *cin* **cmd** *alloc cmd* **ret** *ret*

(*maj***,** *min***,** *chan***) dtr open open mode** *open mode* **ext** *ext* **ret** *ret*

(*maj***,** *min***,** *chan***) dtr open close ret** *ret*

(*maj***,** *min***,** *chan***) dtr open read ret** *ret*

(*maj***,** *min***,** *chan***) dtr open write ret** *ret*

(*maj***,** *min***,** *chan***) dtr open ioctl cmd** *ioctl cmd* **arg** *ioctl arg* **mode** *mode* **ret** *ret*

(*maj***,** *min***,** *chan***) dtr open select events** *events* **revents** *revents* **ret** *ret*

(*maj***,** *min***,** *chan***) dtr open revoke flag** *revoke flag* **ret** *ret*

(*maj***,** *min***,** *chan***) dtr open mpx ret** *ret*

(*maj***,** *min***,** *chan***) dtr open input c** *c input status* **ret** *ret*

(*maj***,** *min***,** *chan***) dtr open output** *output status*

(*maj***,** *min***,** *chan***) dtr open service proc** *proc* **ret** *ret*

(*maj***,** *min***,** *chan***) dtr open service set control** *control* **ret** *ret*

(*maj***,** *min***,** *chan***) dtr open service get control ret** *ret*

(*maj***,** *min***,** *chan***) dtr open service get status ret** *ret*

(*maj***,** *min***,** *chan***) dtr open service baud** *baud* **ret** *ret*

(*maj***,** *min***,** *chan***) dtr open service get baud ret** *ret*

(*maj***,** *min***,** *chan***) dtr open service set input baud** *baud* **ret** *ret*

(*maj***, *min*, *chan*) dtr open service get input baud ret** *ret*

(*maj***, *min*, *chan*) dtr open service set bpc** *bpc* **ret** *ret*

(*maj***, *min*, *chan*) dtr open service get bpc ret** *ret*

(*maj***, *min*, *chan*) dtr open service set parity** *parity* **ret** *ret*

(*maj***, *min*, *chan*) dtr open service get parity ret** *ret*

(*maj***, *min*, *chan*) dtr open service set stops** *stops* **ret** *ret*

(*maj***, *min*, *chan*) dtr open service get stops ret** *ret*

(*maj***, *min*, *chan*) dtr open service set break ret** *ret*

(*maj***, *min*, *chan*) dtr open service clear break ret** *ret*

(*maj***, *min*, *chan*) dtr open service open** *open* **ret** *ret*

(*maj***, *min*, *chan*) dtr open service dopace** *dopace* **ret** *ret*

(*maj***, *min*, *chan*) dtr open service softpace** *softpace* **ret** *ret*

(*maj***, *min*, *chan*) dtr open service softrchar** *softrchar* **ret** *ret*

(*maj***, *min*, *chan*) dtr open service softlchar** *softlchar* **ret** *ret*

(*maj***, *min*, *chan*) dtr open service softrstr** *softrstr* **ret** *ret*

(*maj***, *min*, *chan*) dtr open service softlstr** *softlstr* **ret** *ret*

(*maj***, *min*, *chan*) dtr open service hardrbits** *hardrbits* **ret** *ret*

(*maj***, *min*, *chan*) dtr open service hardlbits** *hardlbits* **ret** *ret*

(*maj***, *min*, *chan*) dtr open service loop enter ret** *ret*

(*maj***, *min*, *chan*) dtr open service loop exit ret** *ret*

(*maj***, *min*, *chan*) dtr open proc** *proc* **ret** *ret*

(*maj***, *min*, *chan*) dtr open slih intr** *intr slih status*

(*maj***, *min*, *chan*) dtr open offlevel intr** *intr* **ret** *ret*

(*maj***, *min*, *chan*)** Major and minor device number, and channel number.

config cmd *cmd*

cin *cin*

cmd *alloc cmd*
Possible values:

push

pop

unconfig

mode *open mode*

ext *ext*

ioctl cmd *ioctl cmd*

arg *ioctl arg*

mode *mode*

events *events*
Possible values:

in

out

pri

sync

revents *revents*

revoke flag *revoke flag*

c *c*

input status
Possible values:

good char

overrun

parity error

framing error

break interrupt

cts on

cts off

dsr on

dsr off

ri on

ri off

cd on

cd off

cblock buf

other buf

proc *proc*	Possible values: **output** **suspend** **resume** **block** **unblock** **rflush** **wflush**
output *output status*	Possible values: **done** **more output**
set control *control*	Possible values: **TSDTR** **TSRTS** **TSCTS** **TSDSR** **TSRI** **TSCD** **baud** *baud* **bpc** *bpc*
parity *parity*	Possible values: **none** **odd** **mark** **even** **space**
stops *stops*	Possible values: **1** **2**
open *open*	Possible values: **local** **remote**

dopace *dopace* Possible values:

again

xon

str

dtr

rts

softpace *softpace* Possible values:

remote off

remote any

remote on

remote str

local off

local on

local str

softrchar *softrchar*

softlchar *softlchar*

softrstr *softrstr*

softlstr *softlstr*

hardrbits *hardrbits*

hardlbits *hardlbits*

intr *intr*

slih status Possible values:

serviced

no intr serviced

ret *ret*

Trace Hook IDs: 411 through 418

The following trace hook IDs are stored in the **/usr/include/sys/trchkid.h** file.

411: HKWD STTY STRTTY

This event is recorded by the tty stream head.

Recorded Data

Events:

(*maj***,** *min***) sth revoke flag** *flag*

(*maj***,** *min***) sth ioctl osr** *osr* **cmd** *ioctl_cmd*

(*maj***,** *min***) sth** *event* **ret** *ret* **from line** *line*

(*maj***,** *min***)**	Major and minor device number.
flag *flag*	Result of a **frevoke** or **revoke** system call.
osr *osr*	Pointer to a structure representing the operating system request.
cmd *ioctl_cmd*	Symbolic name of the ioctl command.
event	One of the recorded event. Possible values: **revoke** **ioctl**
ret *ret*	Function's return value.
from line *line*	Function's return line number.

412: HKWD STTY LDTERM

This event is recorded by the **ldterm** line discipline module.

Recorded Data

(*maj***,** *min***) ldterm config cmd** *cmd*

(*maj***,** *min***) ldterm open ptr** *ptr* **mode:** *mode* **sflag:** *sflag*

(*maj***,** *min***) ldterm close ptr** *ptr* **mode:** *mode*

(*maj***,** *min***) ldterm wput ptr** *ptr* **msg** *msg* **msg_type** *type*

(*maj***,** *min***) ldterm rput ptr** *ptr* **msg** *msg* **msg_type** *type*

(*maj***,** *min***) ldterm wsrv ptr** *ptr* **q_count** *count*

(*maj***,** *min***) ldterm rsrv ptr** *ptr* **q_count** *count*

(*maj***,** *min***) ldterm ioctl ptr** *ptr* **cmd** *ioctl_cmd*

(*maj***,** *min***) ldterm** *event* **ret** *ret* **from line** *line*

(*maj***,** *min***)**	Major and minor device number.
cmd *cmd*	Configuration command. Possible values: **CFG_INIT** **CFG_TERM**
ptr *ptr*	Pointer to the module's private structure (the **ldtty** structure).

mode: *mode*	Open mode of the stream. Possible values:
	READ
	WRITE
	NONBLOCK
	EXCL
	NOCTTY
	NDELAY
sflag: *sflag*	Possible values:
	0
	MODOPEN
	CLONEOPEN
msg *msg*	Message to be processed.
msg_type *type*	Message type. Possible values:
	M_DATA
	M_PROTO
	M_BREAK
	M_SIG
	M_DELAY
	M_CTL
	M_IOCTL
	M_SETOPS
q_count *count*	Total amount of data in the queue.
cmd *ioctl_cmd*	Symbolic name of the ioctl command.
event	One of the recorded event. Possible values:
	config
	open
	close
	wput
	rput
	wsrv
	rsrv
	ioctl
ret *ret*	Function's return value.
from line *line*	Function's return line number.

413: HKWD STTY SPTR

This event is recorded by the **sptr** serial printer module.

Recorded Data

(*maj***,** *min***) sptr config cmd** *cmd*

(*maj***,** *min***) sptr open ptr** *ptr* **mode:** *mode* **sflag:** *sflag*

(*maj***,** *min***) sptr close ptr** *ptr* **mode:** *mode*

(*maj***,** *min***) sptr wput ptr** *ptr* **msg** *msg* **msg_type** *type*

(*maj***,** *min***) sptr rput ptr** *ptr* **msg** *msg* **msg_type** *type*

(*maj***,** *min***) sptr wsrv ptr** *ptr* **q_count** *count*

(*maj***,** *min***) sptr rsrv ptr** *ptr* **q_count** *count*

(*maj***,** *min***) sptr ioctl ptr** *ptr* **cmd** *ioctl_cmd*

(*maj***,** *min***) sptr** *event* **ret** *ret* **from line** *line*

(*maj***,** *min***)**	Major and minor device number.
cmd *cmd*	Configuration command. Possible values:
	CFG_INIT
	CFG_TERM
ptr *ptr*	Pointer to the module's private structure (the **sptr_config** structure).
mode: *mode*	Open mode of the file. Possible values:
	READ
	WRITE
	NONBLOCK
	EXCL
	NOCTTY
	NDELAY
sflag: *sflag*	Possible values:
	0
	MODOPEN
	CLONEOPEN
msg *msg*	Message to be processed.

msg_type *type*	Message type. Possible values:
	M_DATA
	M_PROTO
	M_BREAK
	M_PASSFP
	M_SIG
	M_DELAY
	M_CTL
	M_IOCTL
	M_SETOPS
	M_RSE
q_count *count*	Total amount of data in the queue.
cmd *ioctl_cmd*	Symbolic name of the ioctl command.
event	One of the recorded event. Possible values:
	config
	open
	close
	wput
	rput
	wsrv
	rsrv
	ioctl
ret *ret*	Function's return value.
from line *line*	Function's return line number.

414: HKWD STTY NLS

This event is recorded by the **nls** mapping discipline module.

Recorded Data

(*maj***,** *min***) nls config cmd** *cmd*

(*maj***,** *min***) nls open ptr** *ptr* **mode:** *mode* **sflag:** *sflag*

(*maj***,** *min***) nls close ptr** *ptr* **mode:** *mode*

(*maj***,** *min***) nls wput ptr** *ptr* **msg** *msg* **msg_type** *type*

(*maj***,** *min***) nls rput ptr** *ptr* **msg** *msg* **msg_type** *type*

(*maj***,** *min***) nls wsrv ptr** *ptr* **q_count** *count*

(*maj***,** *min***) nls rsrv ptr** *ptr* **q_count** *count*

(*maj***,** *min***) nls ioctl ptr** *ptr* **cmd** *ioctl_cmd*

(*maj***,** *min***) nls** *event* **ret** *ret* **from line** *line*

(*maj***,** *min***)**	Major and minor device number.
cmd *cmd*	Configuration command. Possible values:
	CFG_INIT
	CFG_TERM
ptr *ptr*	Pointer to the module's private structure (the **nls** structure).
mode: *mode*	Open mode of the file. Possible values:
	READ
	WRITE
	NONBLOCK
	EXCL
	NOCTTY
	NDELAY
sflag: *sflag*	Possible values:
	0
	MODOPEN
	CLONEOPEN
msg *msg*	Message to be processed.
msg_type *type*	Message type. Possible values:
	M_DATA
	M_PROTO
	M_BREAK
	M_PASSFP
	M_SIG
	M_DELAY
	M_CTL
	M_IOCTL
	M_SETOPS
	M_RSE
q_count *count*	Total amount of data in the queue.
cmd *ioctl_cmd*	Symbolic name of the ioctl command.

event One of the recorded event. Possible values:

config

open

close

wput

rput

wsrv

rsrv

ioctl

ret *ret* Function's return value.

from line *line* Function's return line number.

415: HKWD STTY PTY

This event is recorded by the **pty** pseudo-device driver.

Recorded Data

(*maj***,** *min***) pty config cmd** *cmd*

(*maj***,** *min***) pty open ptr** *ptr* **mode:** *mode* **sflag:** *sflag*

(*maj***,** *min***) pty close ptr** *ptr* **mode:** *mode*

(*maj***,** *min***) pty wput ptr** *ptr* **msg** *msg* **msg_type** *type*

(*maj***,** *min***) pty rput ptr** *ptr* **msg** *msg* **msg_type** *type*

(*maj***,** *min***) pty wsrv ptr** *ptr* **q_count** *count*

(*maj***,** *min***) pty rsrv ptr** *ptr* **q_count** *count*

(*maj***,** *min***) pty ioctl ptr** *ptr* **cmd** *ioctl_cmd*

(*maj***,** *min***) pty** *event* **ret** *ret* **from line** *line*

(*maj***,** *min***)** Major and minor device number.

cmd *cmd* Configuration command. Possible values:

CFG_INIT

CFG_TERM

ptr *ptr* Pointer to the module's private structure (the **pty_s** structure).

mode: *mode* Open mode of the file. Possible values:

NONBLOCK

NDELAY

sflag: *sflag* Possible values:

0

MODOPEN

CLONEOPEN

msg *msg* Message to be processed.

msg_type *type* Message type. Possible values:

M_DATA

M_PROTO

M_BREAK

M_SIG

M_CTL

M_IOCTL

M_SETOPS

q_count *count* Total amount of data in the queue.

cmd *ioctl_cmd* Symbolic name of the ioctl command.

event One of the recorded event. Possible values:

config

open

close

wput

rput

wsrv

rsrv

ioctl

ret *ret* Function's return value.

from line *line* Function's return line number.

416: HKWD STTY RS

This event is recorded by the **rs** tty driver.

Recorded Data

(*maj***,** *min***) rs config cmd** *cmd*

(*maj***,** *min***) rs open ptr** *ptr* **mode:** *mode* **sflag:** *sflag*

(*maj***,** *min***) rs close ptr** *ptr* **mode:** *mode*

(*maj***,** *min***) rs wput ptr** *ptr* **msg** *msg* **msg_type** *type*

(*maj***,** *min***) rs rput ptr** *ptr* **msg** *msg* **msg_type** *type*

(*maj***,** *min***) rs wsrv ptr** *ptr* **q_count** *count*

(*maj***,** *min***) rs rsrv ptr** *ptr* **q_count** *count*

(*maj***,** *min***) rs ioctl ptr** *ptr* **cmd** *ioctl_cmd*

(*maj***,** *min***) rs proc ptr** *ptr proc*

(*maj***,** *min***) rs service ptr** *ptr service*

(*maj***,** *min***) rs slih rintr** *rintr* **adap_type** *adap_type*

(*maj***,** *min***) rs offlevel rintr** *rintr*

(*maj***,** *min***) rs** *event* **ret** *ret* **from line** *line*

(*maj***,** *min***)** Major and minor device number.

cmd *cmd* Configuration command. Possible values:

CFG_INIT

CFG_TERM

CFG_QVPD

ptr *ptr* Pointer to the driver's private structure (the **str_rs** structure).

mode: *mode* Open mode of the file. Possible values:

READ

WRITE

NONBLOCK

EXCL

NOCTTY

NDELAY

sflag: *sflag* Possible values:

0

MODOPEN

CLONEOPEN

msg *msg* Message to be processed.

msg_type *type*
Message type. Possible values:

M_DATA

M_PROTO

M_BREAK

M_SIG

M_DELAY

M_CTL

M_IOCTL

q_count *count*
Total amount of data in the queue.

cmd *ioctl_cmd*
Symbolic name of the ioctl command.

proc
Possible values:

output

suspend

resume

block

unblock

rflush

wflush

service Driver internal service. Possible values:

proc output | **suspend** | **resume** | **block** | **unblock** | **rflush** | **wflush**

set control { **TSDTR** | **TSRTS** }

get control

get status

sbaud *baud*

get baud

set input baud *baud*

get input baud

set bpc *bpc*

set parity none | **odd** | **mark** | **even** | **space**

get parity

set stops 1 | **2**

get stops

set break

clear break

open local | **remote**

softpace remote off | **remote any** | **remote on** | **local off** | **local on**

softrchar *char*

softlchar *char*

hardrbits *bits*

hardlbits *bits*

loop enter | **exit**

rintr Pointer to the interupt handler structure.

adap_type Adapter type. Possible values:

Native io

8/16 Port

event	One of the recorded event. Possible values:
	config
	open
	close
	wput
	rput
	wsrv
	rsrv
	ioctl
	service
	slih
	offlevel
ret *ret*	Function's return value.
from line *line*	Function's return line number.

417: HKWD STTY LION

This event is recorded by the **lion** tty driver.

Recorded Data

(*maj***,** *min***) lion config cmd** *cmd*

(*maj***,** *min***) lion open ptr** *ptr* **mode:** *mode* **sflag:** *sflag*

(*maj***,** *min***) lion close ptr** *ptr* **mode:** *mode*

(*maj***,** *min***) lion wput ptr** *ptr* **msg** *msg* **msg_type** *type*

(*maj***,** *min***) lion rput ptr** *ptr* **msg** *msg* **msg_type** *type*

(*maj***,** *min***) lion wsrv ptr** *ptr* **q_count** *count*

(*maj***,** *min***) lion lionrv ptr** *ptr* **q_count** *count*

(*maj***,** *min***) lion ioctl ptr** *ptr* **cmd** *ioctl_cmd*

(*maj***,** *min***) lion proc ptr** *ptr proc*

(*maj***,** *min***) lion service ptr** *ptr service*

(*maj***,** *min***) lion slih rintr** *rintr* **adap_type** *adap_type*

(*maj***,** *min***) lion offlevel rintr** *rintr*

(*maj***,** *min***) lion** *event* **ret** *ret* **from line** *line*

(*maj***,** *min***)**	Major and minor device number.

cmd *cmd* Configuration command. Possible values:

CFG_INIT

CFG_TERM

CFG_QVPD

ptr *ptr* Pointer to the driver's private structure (the **str_lion** structure).

mode: *mode* Open mode of the file. Possible values:

READ

WRITE

NONBLOCK

APPEND

CREAT

TRUNC

EXCL

NOCTTY

NDELAY

sflag: *sflag* Possible values:

0

MODOPEN

CLONEOPEN

msg *msg* Message to be processed.

msg_type *type* Message type. Possible values:

M_DATA

_PROTO

M_BREAK

M_PASSFP

M_SIG

M_DELAY

M_CTL

M_IOCTL

_SETOPS

M_RSE

q_count *count* Total amount of data in the queue.

cmd *ioctl_cmd* Symbolic name of the ioctl command.

proc Possible values:

output

suspend

resume

block

unblock

rflush

wflush

service Driver internal service. Possible values:

proc output | **suspend** | **resume** | **block** | **unblock** | **rflush** | **wflush**

set control { **TSDTR** | **TSRTS** }

get control

get status

sbaud *baud*

get baud

set input baud *baud*

get input baud

set bpc *bpc*

set parity none | **odd** | **mark** | **even** | **space**

get parity

set stops 1 | **2**

get stops

set break

clear break

open local | **remote**

dopace again | **xon** | **str** | **dtr** | **rts**

softpace remote off | **remote any** | **remote on** | **remote str** |

local off | **local on** | **local str**

softrchar *char*

softlchar *char*

softrstr *str*

softlstr *str*

hardrbits *bits*

hardlbits *bits*

loop enter | **exit**

rintr

adap_type Adapter type. Possible value:

64-port adapter

event	One of the recorded event. Possible values:
	config
	open
	close
	wput
	rput
	wsrv
	rsrv
	ioctl
	service
	slih
	offlevel
ret *ret*	Function's return value.
from line *line*	Function's return line number.

418: HKWD STTY CXMA

This event is recorded by the **cxma** tty driver.

Recorded Data

(*maj***,** *min***) cxma config cmd** *cmd*

(*maj***,** *min***) cxma open ptr** *ptr* **mode:** *mode* **sflag:** *sflag*

(*maj***,** *min***) cxma close ptr** *ptr* **mode:** *mode*

(*maj***,** *min***) cxma wput ptr** *ptr* **msg** *msg* **msg_type** *type*

(*maj***,** *min***) cxma rput ptr** *ptr* **msg** *msg* **msg_type** *type*

(*maj***,** *min***) cxma wsrv ptr** *ptr* **q_count** *count*

(*maj***,** *min***) cxma cxmarv ptr** *ptr* **q_count** *count*

(*maj***,** *min***) cxma ioctl ptr** *ptr* **cmd** *ioctl_cmd*

(*maj***,** *min***) cxma proc ptr** *ptr proc*

(*maj***,** *min***) cxma service ptr** *ptr service*

(*maj***,** *min***) cxma slih rintr** *rintr* **adap_type** *adap_type*

(*maj***,** *min***) cxma offlevel rintr** *rintr*

(*maj***,** *min***) cxma** *event* **ret** *ret* **from line** *line*

(*maj***,** *min***)**	Major and minor device number.

cmd *cmd*	Configuration command. Possible values:
	CFG_INIT
	CFG_TERM
	CFG_QVPD
	CFG_UCODE
ptr *ptr*	Pointer to the module's instance structure.
mode: *mode*	Open mode of the file. Possible values:
	READ
	WRITE
	NONBLOCK
	EXCL
	NOCTTY
	NDELAY
sflag: *sflag*	Possible values:
	0
	MODOPEN
	CLONEOPEN
msg *msg*	Message to be processed.
msg_type *type*	Message type. Possible values:
	M_DATA
	M_PROTO
	M_BREAK
	M_PASSFP
	M_SIG
	M_DELAY
	M_CTL
	M_IOCTL
	M_SETOPS
	M_RSE
q_count *count*	Total amount of data in the queue.
cmd *ioctl_cmd*	Symbolic name of the ioctl command.

proc Possible values:

output

suspend

resume

block

unblock

rflush

wflush

service Driver internal service. Possible values:

proc output | **suspend** | **resume** | **block** | **unblock** | **rflush** | **wflush**

set control { **TSDTR** | **TSRTS** | **TSCTS** | **TSDSR** | **TSRI** | **TSCD** }

get control

get status

sbaud *baud*

get baud

set input baud *baud*

get input baud

set bpc *bpc*

set parity none | **odd** | **mark** | **even** | **space**

get parity

set stops 1 | **2**

get stops

set break

clear break

open local | **remote**

dopace again | **xon** | **str** | **dtr** | **rts**

softpace remote off | **remote any** | **remote on** | **remote str** |

local off | **local on** | **local str**

softrchar *char*

softlchar *char*

softrstr *str*

softlstr *str*

hardrbits *bits*

hardlbits *bits*

loop enter | **exit**

rintr

adap_type Possible values:

cxma

event	One of the recorded event. Possible values: **config** **open** **close** **wput** **rput** **wsrv** **rsrv** **ioctl** **service** **slih** **offlevel**
ret *ret*	Function's return value.
from line *line*	Function's return line number.

Trace Hook IDs: 460 through 46E

The following trace hook IDs are stored in the **/usr/include/sys/trchkid.h** file.

460: HKWD KERN ASSERTWAIT

This event is recorded by the **e_assert_wait** kernel service.

Recorded Data

e_assert_wait: tid=*tid* **anchor=***anchor* **flag=***flag* **lr=***lr*

tid	Thread ID of the calling kernel thread.
anchor	The *event_word* parameter; the anchor to the list of kernel threads waiting on this event.
flag	The *interruptible* parameter.
lr	Value of the link register, specifying the return address of the service.

461: HKWD KERN CLEARWAIT

This event is recorded by the **e_clear_wait** kernel service.

Recorded Data

e_clear_wait: tid=*tid* **anchor=***anchor* **result=***result* **lr=***lr*

tid	The *tid* parameter; the thread ID of the kernel thread to be awakened.
anchor	Anchor to the event list where the target thread is sleeping.
result	The *result* parameter; the value to return to the awkened thread.
lr	Value of the link register, specifying the return address of the service.

462: HKWD KERN THREADBLOCK

This event is recorded by the **e_block_thread** kernel service.

Recorded Data

e_block_thread: tid=*tid* **anchor=***anchor* **t_flags=***t_flags* **lr=***lr*

tid	Thread ID of the calling kernel thread.
anchor	Anchor to the event list where the kernel thread will sleep.
t_flags	Flags of the kernel thread.
lr	Value of the link register, specifying the return address of the service.

463: HKWD KERN EMPSLEEP

This event is recorded by the **e_mpsleep** kernel service.

Recorded Data

e_mpsleep: tid=*tid* **anchor=***anchor* **timeout=***timeout* **lock=***lock* **flags=***flags* **lr=***lr*

tid	Thread ID of the calling kernel thread.
anchor	The *event_word* parameter; the anchor to the list of kernel threads waiting on this event.
timeout	The *timeout* parameter; the timeout for the sleep.
lock	The *lock_word* parameter; the lock (simple or complex) to unlock by the kernel service.
flags	The *flags* parameter; the lock and signal handling options.
lr	Value of the link register, specifying the return address of the service.

464: HKWD KERN EWAKEUPONE

This event is recorded by the **e_wakeup_one** kernel service.

Recorded Data

e_wakeup_one: tid=*tid* **anchor=***anchor* **lr=***lr*

tid	Thread ID of the calling kernel thread.
anchor	The *event_word* parameter; the anchor to the list of kernel threads waiting on this event.
lr	Value of the link register, specifying the return address of the service.

465: HKWD SYSC CRTHREAD

This event is recorded by the **thread_create** system call.

Recorded Data

thread_create: pid=*pid* **tid=***tid* **priority=***priority* **policy=***policy*

pid	Process ID of the calling kernel thread's process.
tid	Thread ID of the calling kernel thread.
priority	Priority of the new kernel thread.
policy	Scheduling policy of the new kernel thread.

466: HKWD KERN KTHREADSTART

This event is recorded by the **kthread_start** kernel service.

Recorded Data

kthread_start: pid=*pid* **tid=***tid* **priority=***priority* **policy=***policy* **func=***func*

pid	Process ID of the calling kernel thread's process.
tid	The *tid* parameter; the thread ID of the kernel thread to start.
priority	Priority of the new kernel thread.
policy	Scheduling policy of the new kernel thread.
func	The *i_func* parameter, the address of the new kernel thread's entry-point routine.

467: HKWD SYSC TERMTHREAD

This event is recorded by the **thread_terminate** system call.

Recorded Data

thread_terminate: pid=*pid* **tid=***tid*

pid	Process ID of the calling kernel thread's process.
tid	Thread ID of the calling kernel thread.

468: HKWD KERN KSUSPEND

This event is recorded by the **ksuspend** subroutine. This subroutine is used internally by the system and is undocumented.

Recorded Data

ksuspend: tid=*tid* **p_suspended=***suspended* **p_active=***active*

tid	Thread ID of the calling kernel thread.
suspended	Number of suspended kernel threads in the process.
active	Number of active (suspendable) kernel threads in the process.

469: HKWD SYSC THREADSETSTATE

This event is recorded by the **thread_setstate** system call.

Recorded Data

thread_setstate: tid=*tid* **t_state=***t_state* **t_flags=***t_flags* **priority=***priority* **policy=***policy*

tid	Thread ID of the target kernel thread.
t_state	Current state of the kernel thread. Possible values: **NONE** **IDLE** **RUN** **SLEEP** **SWAP** **STOP** **ZOMB**
t_flags	New flags of the kernel thread.
priority	New priority of the kernel thread.

policy	New scheduling policy of the kernel thread.

46A: HKWD SYSC THREADTERM ACK

This event is recorded by the **thread_terminate_ack** system call.

Recorded Data

thread_terminate_ack: current_tid=*crt_tid* **target_tid=***targ_tid*

crt_tid	Thread ID of the calling kernel thread.
targ_tid	Thread ID of the target kernel thread.

46B: HKWD SYSC THREADSETSCHED

This event is recorded by the **thread_setsched** system call.

Recorded Data

thread_setsched: pid=*pid* **tid=***tid* **priority=***priority* **policy=***policy*

pid	Process ID of the calling kernel thread's process.
tid	The *tid* parameter; the thread ID of the target kernel thread.
priority	The *priority* parameter; the priority to set.
policy	The *policy* parameter; the scheduling policy to set.

46C: HKWD KERN TIDSIG

This event is recorded by the **tidsig** subroutine. This subroutine is used internally by the system and is undocumented.

Recorded Data

tidsig: pid=*pid* **tid=***tid* **signal=***signal* **lr=***lr*

pid	Process ID of the calling kernel thread's process.
tid	Thread ID of the calling kernel thread.
signal	Symbolic name of the delivered signal.
lr	Value of the link register, specifying the return address of the routine.

46D: HKWD KERN WAITLOCK

This event is recorded by the **wait_on_lock** subroutine. This subroutine is used internally by the system and is undocumented.

Recorded Data

wait_on_lock: pid=*pid* **tid=***tid* **lockaddr=***lockaddr*

pid	Process ID of the calling kernel thread's process.
tid	Thread ID of the calling kernel thread.
lockaddr	Address of the lock.

46E: HKWD KERN WAKEUPLOCK

This event is recorded by the **wakeup_lock** subroutine. This subroutine is used internally by the system and is undocumented.

Recorded Data

wakeup_lock: lockaddr=*lockaddr* **waiters=***waiters*

lockaddr	Address of the lock.
waiters	Number of kernel threads remaining sleeping on the lock.

Chapter 28. tty Subsystem

AIX is a multiuser operating system that allows user access from local or remote attached device. The communication layer that supports this function is the tty subsystem.

The communication between terminal devices and the programs that read and write to them is controlled by the tty interface. Examples of tty devices are:

- Modems
- ASCII terminals
- System console
- Serial printer
- System console
- Xterm or aixterm under X-Windows

TTY Subsystem Objectives

The tty subsystem is responsible for:

- Controlling the physical flow of data on asynchronous lines (including the transmission speed, character size, line availability)
- Interpreting the data by recognizing special characters and adapting to national languages
- Controlling jobs and terminal accesses by using the concept of controlling terminal

A controlling terminal manages the input and output operations of a group of processes. The **tty** special file supports the controlling terminal interface. In practice, user programs seldom open terminal files, such as **dev/tty5**. These files are opened by a **getty** or **rlogind** command and become the user's standard input and output devices.

See ″tty Special File″ in *AIX 5L Version 5.1 Files Reference* for more information about controlling terminal.

tty Subsystem Modules

To perform these tasks, the tty subsystem is composed of modules, or disciplines. A module is a set of processing rules that govern the interface for communication between the computer and an asynchronous device. Modules can be added and removed dynamically for each tty.

The tty subsystem supports three main types of modules:

- "tty Drivers"
- "Line Disciplines" on page 880
- "Converter Modules" on page 880

tty Drivers

tty drivers, or hardware disciplines, directly control the hardware (tty devices) or pseudo-hardware (pty devices). They perform the actual input and output to the adapter by providing services to the modules above it: flow control and special semantics when a port is being opened.

The following tty drivers are provided:

cxma 128-port asynchronous controller High-function terminal. The tty name is **/dev/hft/***Y*, where *Y* > = 0.
lft Low-function terminal. The tty name is **/dev/lft***Y*, where *Y* >= 0.
lion 64-port asynchronous controller.

pty	pseudo-terminal device driver.
rs	Native, 8-port, and 16-port asynchronous controller.

The section, “TTY Drivers” on page 887, provides more information.

Line Disciplines

Line disciplines provide editing, job control, and special character interpretation. They perform all transformations that occur on the inbound and outbound data stream. Line disciplines also perform most of the error handling and status monitoring for the tty driver.

The following line disciplines are provided:

ldterm	Terminal devices (see “Line Discipline Module (ldterm)” on page 883)
sptr	Serial printer (**splp** command)
slip	Serial Line Internet Protocol (**slattach** command)

Converter Modules

Converter modules, or *mapping disciplines*, translate, or map, input and output characters.

The following converter modules are provided:

nls	National language support for terminal mapping; this converter translates incoming and outgoing characters on the data stream, based on the input and output maps defined for the port (see the **setmaps** command)
lc_sjis and **uc_sjis**	Upper and lower converter used to translate multibyte characters between the Shifted Japanese Industrial Standard (SJIS) and the Advanced Japanese EUC Code (AJEC) handled by the **ldterm** line discipline.

The section, “Converter Modules” on page 886, provides more information.

TTY Subsystem Structure

The tty subsystem is based on STREAMS. This STREAMS-based structure provides modularity and flexibility, and enables the following features:

- Easy customizing; users can customize their terminal subsystem environment by adding and removing modules of their choice.
- Reusable modules; for example, the same line discipline module can be used on many tty devices with different configurations.
- Easy addition of new features to the terminal subsystem.
- Providing an homogeneous tty interface on heterogeneous devices.

The structure of a tty stream is made up of the following modules:

- The stream head, processing the user’s requests. The stream head is the same for all tty devices, regardless of what line discipline or tty driver is in use.
- An optional upper converter (**uc_sjis** for example), a converter module pushed above the line discipline to convert upstream and downstream data.
- The line discipline.
- An optional lower converter (**lc_sjis** for example), a converter module pushed below the line discipline to convert upstream and downstream data.
- An optional character mapping module (**nls**), a converter module pushed above the tty driver to support input and output terminal mapping.

- The stream end: a tty driver.

Unless required, the internationalization modules are not present in the tty stream.

For a serial printer, the internationalization modules are usually not present on the stream; therefore, the structure is simpler.

Common Services

The **/usr/include/sys/ioctl.h** and **/usr/include/termios.h** files describe the interface to the common services provided by the tty subsystem. The **ioctl.h** file, which is used by all of the modules, includes the **winsize** structure, as well as several ioctl commands. The **termios.h** file includes the POSIX compliant subroutines and data types.

The provided services are grouped and discussed here according to their specific functions.

- "Hardware Control Services":
  - **cfgetispeed** subroutine
  - **cfgetospeed** subroutine
  - **cfsetispeed** subroutine
  - **cfsetospeed** subroutine
  - **tcsendbreak** subroutine
- "Flow Control Services" on page 882:
  - **tcdrain** subroutine
  - **tcflow** subroutine
  - **tcflush** subroutine
- "Terminal Information and Control" on page 882:
  - **isatty** subroutine
  - **tcgetattr** subroutine
  - **tcsetattr** subroutine
  - **ttylock** subroutine
  - **ttylocked** subroutine
  - **ttyname** subroutine
  - **ttyunlock** subroutine
  - **ttywait** subroutine
- "Window and Terminal Size Services" on page 882:
  - **termdef** subroutine
  - **TIOCGWINSZ** ioctl operation
  - **TIOCSWINSZ** ioctl operation
- "Process Group Management Services" on page 882:
  - **tcgetpgrp** subroutine
  - **tcsetpgrp** subroutine

Hardware Control Services

The following subroutines are provided for hardware control:

cfgetispeed	Gets input baud rate
cfgetospeed	Gets output baud rate
cfsetispeed	Sets input baud rate
cfsetospeed	Sets output baud rate
tcsendbreak	Sends a break on an asynchronous serial data line

Flow Control Services

The following subroutines are provided for flow control:

tcdrain	Waits for output to complete
tcflow	Performs flow control functions
tcflush	Discards data from the specified queue

Terminal Information and Control

The following subroutines are provided for terminal information and control:

isatty	Determines if the device is a terminal
setcsmap	Reads a code set map file and assigns it to the standard input device
tcgetattr	Gets terminal state
tcsetattr	Sets terminal state
ttylock, **ttywait**, **ttyunlock**, or **ttylocked**	Controls tty locking functions
ttyname	Gets the name of a terminal

Window and Terminal Size Services

The kernel stores the **winsize** structure to provide a consistent interface for the current terminal or window size. The **winsize** structure contains the following fields:

`ws_row`	Indicates the number of rows (in characters) on the window or terminal
`ws_col`	Indicates the number of columns (in characters) on the window or terminal
`ws_xpixel`	Indicates the horizontal size (in pixels) of the window or terminal
`ws_ypixel`	Indicates the vertical size (in pixels) of the window or terminal

By convention, a value of 0 in all of the **winsize** structure fields indicates that the structure has not yet been set up.

termdef	Queries terminal characteristics.
TIOCGWINSZ	Gets the window size. The argument to this ioctl operation is a pointer to a **winsize** structure, into which the current terminal or window size is placed.
TIOCSWINSZ	Sets the window size. The argument to this ioctl operation is a pointer to a **winsize** structure, which is used to set the current terminal or window size information. If the new information differs from the previous, a **SIGWINCH** signal is sent to the terminal process group.

Process Group Management Services

The following subroutines are provided for process group management:

tcgetpgrp	Gets foreground process group ID
tcsetpgrp	Sets foreground process group ID

Buffer Size Operations

The following ioctl operations are used for setting the size of the terminal input and output buffers. The argument to these operations is a pointer to an integer specifying the size of the buffer.

TXSETIHOG	Sets the hog limit for the number of input characters that can be received and stored in the internal tty buffers before the process reads them. The default hog limit is 512 characters. Once the hog limit plus one character is reached, an error is logged in the error log and the input buffer is flushed. The hog number should not be too large, since the buffer is allocated from the system-pinned memory.

TXSETOHOG Sets the hog limit for the number of output characters buffered to echo input. The default hog limit is 512 characters. Once the hog output limit is reached, input characters are no longer echoed. The hog number should not be too large, since the buffer is allocated from the system-pinned memory.

Synchronization

The tty subsystem takes advantage of the synchronization provided by STREAMS. The tty stream modules are configured with the queue pair level synchronization. This synchronization allows the parallelization of the processing for two different streams.

Line Discipline Module (ldterm)

The **ldterm** line discipline is the common line discipline for terminals. This line discipline is POSIX compliant and also ensures compatibility with the BSD interface. The latter line discipline is supported only for compatibility with older applications. For portability reasons, it is strongly recommended that you use the POSIX line discipline in new applications.

This section describes the features provided by the **ldterm** line discipline. For more information about controlling **ldterm**, see ″termios.h File″ in *AIX 5L Version 5.1 Files Reference*

Terminal Parameters

The parameters that control certain terminal I/O characteristics are specified in the **termios** structure as defined in the **termios.h** file. The **termios** structure includes (but is not limited to) the following members:

`tcflag_t c_iflag`	Input modes
`tcflag_t c_oflag`	Output modes
`tcflag_t c_cflag`	Control modes
`tcflag_t c_lflag`	Local modes
`cc_t c_cc[NCCS]`	Control characters.

The `tcflag_t` and `cc_t` unsigned integer types are defined in the **termios.h** file. The **NCCS** symbol is also defined in the **termios.h** file.

Process Group Session Management (Job Control)

A controlling terminal distinguishes one process group in the session, with which it is associated, to be the foreground process group. All other process groups in the session are designated as background process groups. The foreground process group plays a special role in handling signals.

Command interpreter processes that support job control, such as the Korn shell (the **ksh** command) and the C shell (the **csh** command), can allocate the terminal to different *jobs*, or process groups, by placing related processes in a single process group and associating this process group with the terminal. A terminal's foreground process group can be set or examined by a process, assuming the permission requirements are met. The terminal driver assists in job allocation by restricting access to the terminal by processes that are not in the foreground process group.

Terminal Access Control

If a process that is not in the foreground process group of its controlling terminal attempts to read from the controlling terminal, the process group of that process is sent a **SIGTTIN** signal. However, if the reading process is ignoring or blocking the **SIGTTIN** signal, or if the process group of the reading process is orphaned, the read request returns a value of -1, sets the **errno** global variable to **EIO**, and does not send a signal.

If a process that is not in the foreground process group of its controlling terminal attempts to write to the controlling terminal, the process group of that process is sent a **SIGTTOU** signal. However, the management of the **SIGTTOU** signal depends on the **TOSTOP** flag which is defined in the `c_lflag` field of the **termios** structure. If the **TOSTOP** flag is not set, or if the **TOSTOP** flag is set and the process is ignoring or blocking the **SIGTTOU** signal, the process is allowed to write to the terminal, and the **SIGTTOU** signal is not sent. If the **TOSTOP** flag is set, the process group of the writing process is orphaned, and the writing process is not ignoring or blocking the **SIGTTOU** signal, then the write request returns a value of -1, sets the **errno** global variable to **EIO**, and does not send a signal.

Certain functions that set terminal parameters (**tcsetattr**, **tcsendbreak**, **tcflow**, and **tcflush**) are treated in the same manner as write requests, except that the **TOSTOP** flag is ignored. The effect is identical to that of terminal write requests when the **TOSTOP** flag is set.

Reading Data and Input Processing

For terminals that operate in full-duplex mode, data can arrive even while output is occurring. Each terminal device file is associated with an *input queue*, where incoming data is stored by the system before being read by a process. The system imposes a limit (defined by the **MAX_INPUT** constant in the **limits.h** header file) on the number of bytes that can be stored in the input queue. When the input limit is reached, all the saved characters are thrown away without notice.

Two general kinds of input processing are available, depending on whether the terminal device file is in canonical or noncanonical mode. Additionally, input characters are processed according to the `c_iflag` and `c_lflag` fields. Such processing can include *echoing*, or the transmitting of input characters immediately back to the terminal that sent them. Echoing is useful for terminals that can operate in full-duplex mode.

A read request can be handled in two ways, depending on whether the **O_NONBLOCK** flag is set by an **open** or **fcntl** subroutine. If the **O_NONBLOCK** flag is not set, the read request is blocked until data is available or until a signal is received. If the **O_NONBLOCK** flag is set, the read request is completed, without blocking, in one of three ways:

- If there is enough data available to satisfy the entire request, the read request completes successfully and returns the number of bytes read.
- If there is not enough data available to satisfy the entire request, the read request completes successfully, having read as much data as possible, and returns the number of bytes it was able to read.
- If there is no data available, the read request returns a value of -1 and sets the **errno** global variable to **EAGAIN**.

The availability of data depends on whether the input processing mode is canonical or noncanonical. The canonical or noncanonical modes can be set with the **stty** command.

Canonical Mode Input Processing

In canonical mode input processing (**ICANON** flag set in `c_lflag` field of **termios** structure), terminal input is processed in units of lines. A line is delimited by a new-line (ASCII LF) character, an end-of-file (EOF) character, or an end-of-line (EOL) character. This means that a program attempting to read is blocked until an entire line has been typed or a signal has been received. Also, regardless of how many characters are specified in the read request, no more than one line is returned. It is not, however, necessary to read an entire line at once. Any number of characters can be specified in a read request without losing information. During input, erase and kill processing is done.

ERASE character	(Backspace, by default) erases the last character typed
WERASE character	(Ctrl-W key sequence, by default) erases the last word typed in the current line, but not any preceding spaces or tabs

(A *word* is defined as a sequence of nonblank characters; tabs are regarded as blanks.) Neither the ERASE nor the WERASE character erases beyond the beginning of the line.

KILL character	(Ctrl-U sequence, by default) deletes the entire input line and, optionally, outputs a new-line character

All of these characters operate on a keystroke basis, independent of any backspacing or tabbing that might have been done.

REPRINT character	(Ctrl-R sequence, by default) prints a new line followed by the characters from the previous line that have not been read

Reprinting also occurs automatically if characters that would normally be erased from the screen are fouled by program output. The characters are reprinted as if they were being echoed. Consequently, if the **ECHO** flag is not set in the `c_lflag` field of the **termios** structure, the characters are not printed. The ERASE and KILL characters can be entered literally by preceding them with the escape character \ (backslash), in which case, the escape character is not read. The ERASE, WERASE, and KILL characters can be changed.

Noncanonical Mode Input Processing

In noncanonical mode input processing (**-ICANON** flag set in `c_lflag` field of **termios** structure), input bytes are not assembled into lines, and erase and kill processing does not occur.

MIN	Represents the minimum number of bytes that should be received when the read request is successful
TIME	A timer of 0.1-second granularity that is used to time-out burst and short-term data transmissions

The values of the MIN and TIME members of the `c_cc` array are used to determine how to process the bytes received. The MIN and TIME values can be set with the **stty** command. If the MIN value is greater than the defined value of the **MAX_INPUT** constant, the response to the request is implementation-defined. The four possible values for MIN and TIME and their interactions are described in the subsequent paragraphs.

Case A: MIN 0, TIME 0: In this case, TIME serves as an interbyte timer, which is activated after the first byte is received and reset each time a byte is received. If MIN bytes are received before the interbyte timer expires, the read request is satisfied. If the timer expires before MIN bytes are received, the characters received to that point are returned to the user. If TIME expires, at least one byte is returned. (The timer would not have been enabled unless a byte was received.) The read operation blocks until the MIN and TIME mechanisms are activated by the receipt of the first byte or until a signal is received.

Case B: MIN 0, TIME = 0: In this case, only MIN is significant; the timer is not significant (the value of TIME is 0). A pending read request is not satisfied (blocks) until MIN bytes are received or until a signal is received. A program that uses this case to read record-based terminal I/O can block indefinitely in the read operation.

Case C: MIN = 0, TIME 0: In this case, because the value of MIN is 0, TIME no longer represents an interbyte timer. It now serves as a read timer that is activated as soon as the read request is processed. A read request is satisfied as soon as a byte is received or when the read timer expires. Note that if the timer expires, no bytes are returned. If the timer does not expire, the read request can be satisfied only if a byte is received. In this case, the read operation does not block indefinitely, waiting for a byte. If, after the read request is initiated, no byte is received within the period specified by TIME multiplied by 0.1 seconds, the read request returns a value of 0, having read no data.

Case D: MIN = 0, TIME = 0: In this case, the minimum of either the number of bytes requested or the number of bytes currently available is returned without waiting for more bytes to be input. If no characters are available, the read request returns a value of 0, having read no data.

Cases A and B exist to handle burst-mode activity, such as file transfer programs, where a program needs to process at least the number of characters specified by the MIN variable at one time. In Case A, the interbyte timer is activated as a safety measure. In Case B, the timer is turned off.

Cases C and D exist to handle single-character, limited transfers. These cases are readily adaptable to screen-based applications that need to know if a character is present in the input queue before refreshing the screen. In Case C, the timer is activated. In Case D, the timer is turned off. Case D leads to bad performance; but it is better to use it than doing a read request with setting the **O_NONBLOCK** flag.

Writing Data and Output Processing

When one or more characters are written, they are transmitted to the terminal as soon as previously written characters are displayed. (Input characters are echoed by putting them into the output queue as they arrive.) If a process produces characters more rapidly than they can be displayed, the process is suspended when its output queue exceeds a certain limit. When the queue has drained down to a certain threshold, the program is resumed.

Modem Management

If the **CLOCAL** flag is set in the `c_cflag` field of the **termios** structure, a connection does not depend on the state of the modem status lines. If the **CLOCAL** flag is clear, the modem status lines are monitored. Under normal circumstances, an **open** function waits for the modem connection to complete. However, if the **O_NONBLOCK** or **CLOCAL** flag is set, the open function returns immediately without waiting for the connection.

If the **CLOCAL** flag is not set in the `c_cflag` field of the **termios** structure and a modem disconnect is detected by the terminal interface for a controlling terminal, the **SIGHUP** signal is sent to the controlling process associated with the terminal. Unless other arrangements have been made, this signal causes the process to terminate. If the **SIGHUP** signal is ignored or caught, any subsequent read request returns an end-of-file indication until the terminal is closed. Any subsequent write request to the terminal returns a value of -1 and sets the **errno** global variable to **EIO** until the device is closed.

Closing a Terminal Device File

The last process to close a terminal device file causes any output to be sent to the device and any input to be discarded. Then, if the **HUPCL** flag is set in the `c_cflag` field of the **termios** structure and the communications port supports a disconnect function, the terminal device performs a disconnect.

Converter Modules

Converter modules are optional modules; they are pushed onto a tty stream only if required. They are usually provided for internationalization purposes and perform various character mapping.

The following converter modules are shipped with the Base Operating System:

- The **nls** module
- The **uc_sjis** and **lc_sjis** modules.

NLS Module

The **nls** module is a lower converter module that can be pushed onto a tty stream below the line discipline. The **nls** module ensures terminal mapping: it executes the mapping of input and output characters for nonstandard terminals (that is, for terminals that do not support the basic codeset ISO 8859 of the system).

The mapping rules are specified in two map files located in the **/usr/lib/nls/termmap** directory. The **.in** files contain the mapping rules for the keyboard inputs. The **.out** files contain the mapping rules for the display outputs. The files format is specified in the **setmaps** file format ″setmaps File Format″ in *AIX 5L Version 5.1 Files Reference*.

SJIS Modules

The **uc_sjis** and **lc_sjis** modules are converter modules that can be pushed onto a tty stream. They ensure codeset handling: they execute the conversion of multibyte characters between the shifted Japanese industrial standard (SJIS) format and the advanced Japanese EUC code (AJEC) format, supported by the line disciplines. They are needed when the user process and the hardware terminal uses the IBM-943 or IBM-932 code set.

AJEC is a Japanese implementation of the extended UNIX code (EUC) encoding method, which allows combination of ASCII, phonetic Kana, and ideographic Kanji characters. AJEC is a superset of UNIX Japanese industrial standard (UJIS), a common Japanese implementation of EUC.

Japanese-encoded data consist of characters from up to four code sets:

Code set	Contained characters
ASCII	Roman letters, digits, punctuation and control characters
JIS X0201	Phonetic Kana
JIS X0208	Ideographic Kanji
JIS X0212	Supplemental Kanji.

AJEC makes use of all four code sets. SJIS makes use only of ASCII, JIS X0201, and JIS X0208 code sets. Therefore, the **uc_sjis** and **lc_sjis** modules convert:

- All SJIS characters into AJEC characters
- AJEC characters from ASCII, JIS X0201, and JIS X0208 code sets into SJIS characters
- AJEC characters from JIS X0212 code set into the SJIS undefined character.

The **uc_sjis** and **lc_sjis** modules are always used together. The **uc_sjis** upper converter is pushed onto the tty stream above the line discipline; the **lc_sjis** lower converter is pushed onto the stream below the line discipline. The **uc_sjis** and **lc_sjis** modules are automatically pushed onto the tty stream by the **setmaps** command and the **setcsmap** subroutine. They are also controlled by the EUC ioctl operations described in the **eucioctl.h** file in the *AIX Version 4 Files Reference .*

Related Information

TTY Drivers

A tty driver is a STREAMS driver managing the actual connection to the hardware terminal. Depending on the connection, three kinds of tty drivers are provided: asynchronous line drivers, the pty driver, and the LFT driver.

Asynchronous Line Drivers

The asynchronous line drivers are provided to support devices (usually ASCII terminals) directly connected to the system through asynchronous lines, including modems.

The asynchronous line drivers provide the interface to the line control hardware:

- The **cxma** driver supports the 128-port adapter card.
- The **lion** driver supports the 64-port adapter card.
- The **rs** driver supports the native ports and the 8-port and 16-port adapter cards.

The asynchronous line drivers are responsible for setting parameters, such as the baud rate, the character size, and the parity checking. The user can control these parameters through the `c_cflag` field of the **termios** structure.

The asynchronous line drivers also provide the following features:

- The hardware and software flow control, or pacing discipline, specifies how the connection is managed to prevent a buffer overflow. The user can control this feature through the `c_iflag` field of the **termios** structure (software flow control) and the `x_hflag` field of the **termiox** structure (hardware flow control).
- The open discipline specifies how to establish a connection. This feature is controlled at configuration time through the `x_sflag` field of the **termiox** structure.

Pseudo-Terminal Driver

The pseudo-terminal (pty) driver is provided to support terminals that need special processing, such as X terminals or remote systems connected through a network.

A pty driver just transmits the input and output data from the application to a server process through a second stream. The server process, running in the user space, is usually a daemon, such as the **rlogind** daemon or the **xdm** daemon. It manages the actual communication with the terminal.

Other optional modules may be pushed on either user or server stream.

Related Information

STREAMS Overview in *AIX 5L Version 5.1 Communications Programming Concepts*The **setmaps** command, **stty** command in *AIX 5L Version 5.1 Commands Reference*

The **setmaps** file format, **lp** special file, **pty** special file, **tty** special file, **eucioctl.h** file, **termios.h** file, **termiox.h** file in *AIX 5L Version 5.1 Files Reference*

Chapter 29. High-Resolution Time Measurements Using POWER-based Time Base or POWER family Real-Time Clock

The POWER family and PowerPC 601 RISC Microprocessor have real-time clock registers that can be used to make high-resolution time measurements. These registers provide seconds and nanoseconds.

POWER-based processors other than PowerPC 601 RISC Microprocessor do not have real time clock registers. Instead these processors have a time base register, which is a free-running 64-bit register that increments at a constant rate. The time base register can also be used to make high resolution elapsed-time measurements, but requires calculations to convert the value in the time base register to seconds and nanoseconds.

If an application tries to read the real-time clock registers on a POWER-based processor that does not implement them, the processor generates a trap that causes the instruction to be emulated. The answer is correct, but the emulation requires a much larger number of cycles than just reading the register. If the application uses this for high-resolution timing, the time associated with the emulation is included.

If an application tries to read the time base registers on any processor that does not implement them, including the PowerPC 601 RISC Microprocessor, this will not be emulated and will result in the application being killed.

Beginning with AIX Version 4, the operating system provides the following library services to support writing processor-independent code to perform high resolution time measurements:

read_real_time	Reads either the real-time clock registers or the time base register and stores the result
time_base_to_time	Converts the results of **read_real_time** to seconds and nanoseconds

read_real_time does no conversions and runs quickly. **time_base_to_time** does whatever conversions are needed, based on whether the processor has a real-time clock register or a time base register, to convert the results to seconds and nanoseconds.

Since there are separate routines to read the registers and do the conversions, an application can move the conversion out of time-critical code paths.

Chapter 30. Loader Domains

In some programming environments, it is desirable to have shared libraries loaded at the same virtual address in each process. Due to the dynamic nature of shared libraries maintained by the AIX system loader, this condition cannot be guaranteed. Loader domains provide a means of loading shared libraries at the same virtual address in a set of processes.

The system loader loads shared libraries into multiple global shared library regions. One region is called the shared library text region, which contains the executable instructions for loaded shared libraries. The shared library text region is mapped to the same virtual address in every process. The other region is the shared library data region. This region contains the data for shared libraries. Because shared library data is read/write, each process has its own private region that is a copy of the global shared library region. This private region is mapped to the same virtual address in every process.

Since the global shared library regions are mapped at the same virtual address in every process, shared libraries are loaded at the same virtual address in most cases. The case where this is not true is when there is more than one version of a shared library loaded in the system. This happens whenever a shared library that is in use is modified, or any shared libraries it depends on are modified. When this happens, the loader must create a new version of the modified shared library and all other shared libraries that depend on the modified shared library. Note that all shared libraries ultimately depend on the *Kernel Name Space*. The *Kernel Name Space* contains all the system calls defined by the kernel and can be modified any time a kernel extension is dynamically loaded or unloaded. When the system loader creates a new version of a shared library, the new version must be located at a different location in the global shared library segments. Therefore, processes that use the new version have the shared libraries loaded at a different virtual address than processes that use the previous versions of the shared libraries.

A loader domain is a subset of all the shared libraries that are loaded in the system. The set of all shared libraries loaded in the system is called the *global loader domain*. This global loader domain can be subdivided into smaller user-defined loader domains. A user-defined loader domain contains one version of any particular shared library. Processes can specify a loader domain. If a process specifies a loader domain, the process uses the shared libraries contained in the loader domain. If more than one process specifies the same loader domain, they use the same set of shared libraries. Since a loader domain contains one version of any particular shared library, all processes that specify the same loader domain use the same version of shared libraries and have their shared libraries loaded at the same virtual address.

Using Loader Domains

If a process uses a loader domain, it must be specified at exec time. The loader domain specified is in effect and used for the entire duration of the process. When a process that specifies a loader domain calls the **exec** system call, the system loader takes the following actions:

Finds/creates loader domain

The access permissions associated with the loader domain are checked to determine if this process can use the loader domain. If the process does not have sufficient privilege to access (read or write) the loader domain, no domain is used by the process. If the process does have sufficient privilege, the list of loader domains maintained by the system loader is searched for the loader domain specified by the process. If the loader domain specified is not found, it is created if the process has sufficient privilege. If the process does not have sufficient privilege to create the loader domain, then the **exec** call fails, and an error is returned.

Uses loader domain to limit search

If the process needs any shared libraries that are already listed in the loader domain, the version of the library specified in the domain is used. The version of the shared library in the loader domain is used regardless of other versions of the shared library that may exist in the global loader domain.

Adds shared libraries to loader domain

If the process needs a library that is not in the loader domain, the loader loads the library into the process image by following the normal loader convention of loading the most recent version. If the process has sufficient privilege, this version of the library is also added to the loader domain. If the process does not have sufficient privilege to add an entry, the **exec** call fails, and an error is returned.

Shared libraries can also be explicitly loaded with the **load( )** system call. When a shared library is explicitly loaded, the data for these modules is normally put at the current break value of the process for a 32-bit process. For a 64-bit process, the data for the modules is put in the region's privately loaded modules. If a process uses a loader domain, the system loader puts the data in the shared library data region. The virtual address of this explicitly loaded module is the same for all processes that load the module. If the process has sufficient privilege, the shared library is added to the loader domain. If the process does not have sufficient privilege to add an entry, the **load** call fails, and an error is returned.

A loader domain can be associated with any regular file. It is important to note that a loader domain is associated with the file, NOT the path name of the file. The mode (access permissions) of the file determines the operations that can be performed on the loader domain. Access permissions on the file associated with the loader domain and the operations allowed on the loader domain are as follows:

- If the process is able to read the file, the process can specify the loader domain to limit the set of shared libraries it uses.
- If the process is able to write to the file, the process is able to add shared libraries to the loader domain and create the loader domain associated with the file.

If a process attempts to create or add entires to a loader domain without sufficient privilege, the operation in progress (**exec** or **load**) fails, and an error is returned.

Loader domains are specified as part of the **LIBPATH** information. **LIBPATH** information is a colon (:) separated list of directory path names used to locate shared libraries. **LIBPATH** information can come from either the **LIBPATH** environment variable or the **LIBPATH** string specified in the loader section of the executable file. If the first path name in the **LIBPATH** information is a regular file, a loader domain associated with the file is specified. For example:

- If `/etc/loader_domain/OOdomain_1` is a regular file, then setting the **LIBPATH** environment variable to the string

  ```
  /etc/loader_domain/OOdomain_1:/lib:/usr/lib
  ```

  causes processes to create and use the loader domain associated with the `/etc/loader_domain/OOdomain_1` file.
- If `/etc/loader_domain/OOdomain_1` is a regular file, then the `ldom` program is built with the following command:

  ```
  cc -o ldom ldom.c -L/etc/loader_domain/OOdomain_1
  ```

  The path name `/etc/loader_domain/OOdomain_1` is inserted as the first entry in the **LIBPATH** information of the loader section for the `ldom` file. When `ldom` is executed, it creates and uses the loader domain associated with the `/etc/loader_domain/OOdomain_1` file.

Creating/Deleting Loader Domains

A loader domain is created the first time a process with sufficient privilege attempts to use the domain. Access to a loader domain is controlled by access to the regular file associated with the domain. Application writers are responsible for managing the regular files associated with loader domains used by their applications. Loader domains are associated with regular files NOT the path names of the files. The following examples illustrate this point:

- The `apl` application has specified loader domain `domain01` in its **LIBPATH** information. The `apl` application is then executed. The current working directory is `/home/user1`, and it contains a regular file `domain1` that is writable by `apl`. A new loader domain associated with the file `/home/user1/domain01` is created. `apl` is executed again. This time `/home/user2` is the current working directory, and it also contains a regular file `domain01` that is writable by `apl`. A new loader domain associated with the file `/home/user1/domain02` is created.
- Application `apl` has specified loader domain `/etc/1_domain/domain01` in its **LIBPATH** information. `apl` is then executed. `/etc/1_domain/domain01` is a regular file that is writable by `apl`. A new loader domain associated with the file `/etc/1_domain/domain01` is created.

  `/home/user1/my_domain` is a symbolic link to file `/etc/1_domain/domain01`.

  Application `ap2` has specified loader domain `/home/user1/my_domain` in its **LIBPATH** information. `ap2` is then executed. The system loader notices that `/home/user1/my_domain` refers to the same file as `/etc/1_domain/domain01`. A loader domain is already associated with file `/etc/1_domain/domain01`; therefore, this loader domain is used by application `ap2`.
- Application `apl` has specified loader domain `/etc/1_domain/domain01` in its **LIBPATH** information. `apl` is then executed. `/etc/1_domain/domain01` is a regular file that is writable by `apl`. A new loader domain associated with the file `/etc/1_domain/domain01` is created.

  File `/etc/1_domain/domain01` is deleted and recreated as a regular file.

  Application `apl` is executed again. There is no longer any way to access the regular file that is associated with the original loader domain `/etc/1_domain/domain01`. Therefore, a new loader domain associated with the file `/etc/1_domain/domain01` is created.

Loader domains are dynamic structures. During the life of a loader domain, shared libraries are added and deleted. A shared library is added to a loader domain when a process that specified the loader domain needs a shared library that does not already exist in the domain. Of course, this assumes the process has sufficient privilege to add the shared library to the loader domain.

A separate use count is kept for each shared library that is a member of a loader domain. This use count keeps track of how many processes with loader domains are using the shared library. When this use count drops to zero, the shared library is deleted from the loader domain.

Chapter 31. Power Management-Aware Application Program

Power Management (PM) is a technique that uses hardware and software to minimize system power consumption. Power Management has been primarily available on mobile computer systems, but it is now available in the desktop computer environment. Since PM state transitions introduce different environments to application programs, it may be necessary for some application programs to know the PM state transitions. For example, since all processes are frozen at the suspend or hibernation states, some processes might want to save data in a file before entering suspend or hibernation. Also, some application programs might need to control PM state transitions.

Although the AIX PM system consists of several components, such as the PM core (kernel extension), **PM** daemon, PM system calls, and PM commands, all the communication between the PM system and application programs can be done through the PM library. The PM library provides the following functions to PM-aware application programs:

- Controlling or querying PM parameters
- Controlling or querying PM states
- Querying PM events
- Controlling or querying battery information

Although a PM-aware application can run on any platform, it will only be beneficial on a platform that supports PM.

For detailed information on the syntax and return codes of each PM library function, see *AIX 5L Version 5.1 Technical Reference: Kernel and Subsystems Volume 1*.

Chapter 32. ELF Object Files and Dynamic Linking

These sections contains general information for ELF (Executable and Linking Format) object files and dynamic linking.

Section 1. ELF Object File General Information

ELF Object File General Information

This section describes the object file format, called ELF (Executable and Linking Format). There are three main types of object files.

- A *relocatable file* holds code and data suitable for linking with other object files to create an executable or a shared object file.
- An *executable file* holds a program suitable for execution; the file specifies how **exec**(base operating system) creates a program's process image.
- A *shared object file* holds code and data suitable for linking in two contexts. First, the link editor [see **ld**(base operating system)] processes the shared object file with other relocatable and shared object files to create another object file. Second, the dynamic linker combines it with an executable file and other shared objects to create a process image.

Created by the assembler and link editor, object files are binary representations of programs intended to be executed directly on a processor. Programs that require other abstract machines, such as shell scripts, are excluded.

After the introductory material, this chapter focuses on the file format and how it pertains to building programs. Chapter 32 also describes parts of the object file, concentrating on the information necessary to execute a program.

File Format

Object files participate in program linking (building a program) and program execution (running a program). For convenience and efficiency, the object file format provides parallel views of a file's contents, reflecting the differing needs of those activities. The table below shows an object file's organization.

An *ELF header* resides at the beginning and holds a road map describing the file's organization. *Sections* hold the bulk of object file information for the linking view: instructions, data, symbol table, relocation information, and so on. Descriptions of special sections appear later in the chapter. Chapter 32 discusses *segments* and the program execution view of the file.

A *program header table* tells the system how to create a process image. Files used to build a process image (execute a program) must have a program header table; relocatable files do not need one. A *section header table* contains information describing the file's sections. Every section has an entry in the table; each entry gives information such as the section name, the section size, and so on. Files used during linking must have a section header table; other object files may or may not have one.

> NOTE: Although the figure shows the program header table immediately after the ELF header, and the section header table following the sections, actual files may differ. Moreover, sections and segments have no specified order. Only the ELF header has a fixed position in the file.

Data Representation

As described here, the object file format supports various processors with 8-bit bytes and either 32-bit or 64-bit architectures. Nevertheless, it is intended to be extensible to larger (or smaller) architectures. Object files therefore represent some control data with a machine-independent format, making it possible to identify object files and interpret their contents in a common way. Remaining data in an object file use the encoding of the target processor, regardless of the machine on which the file was created.

32-Bit Data Types

Name	Size	Alignment	Purpose
Elf32_Addr	**4**	**4**	Unsigned program address
Elf32_Off	**4**	**4**	Unsigned file offset
Elf32_Half	**2**	**2**	Unsigned medium integer
Elf32_Word	**4**	**4**	Unsigned integer
Elf32_Sword	**4**	**4**	Signed integer
unsigned char	**1**	**1**	Unsigned small integer

64-Bit Data Types

Name	Size	Alignment	Purpose
Elf64_Addr	**8**	**8**	Unsigned program address
Elf64_Off	**8**	**8**	Unsigned file offset
Elf64_Half	**2**	**2**	Unsigned medium integer
Elf64_Word	**4**	**4**	Unsigned integer
Elf64_Sword	**4**	**4**	Signed integer
Elf64_Xword	**8**	**8**	Unsigned long integer
Elf64_Sxword	**8**	**8**	Signed long integer
unsigned char	**1**	**1**	Unsigned small integer

All data structures that the object file format defines follow the natural size and alignment guidelines for the relevant class. If necessary, data structures contain explicit padding to ensure 8-byte alignment for 8-byte objects, 4-byte alignment for 4-byte objects, to force structure sizes to a multiple of 4 or 8, and so forth.

Data also have suitable alignment from the beginning of the file. Thus, for example, a structure containing an **Elf32_Addr** member will be aligned on a 4-byte boundary within the file.

For portability reasons, ELF uses no bit-fields.

ELF Header

Some object file control structures can grow, because the ELF header contains their actual sizes. If the object file format changes, a program may encounter control structures that are larger or smaller than expected. Programs might therefore ignore extra information. The treatment of missing information depends on context and will be specified when and if extensions are defined.

ELF Header

```
#define EI_NIDENT 16
typedef struct {
        unsigned char   e_ident[EI_NIDENT];
        Elf32_Half      e_type;
        Elf32_Half      e_machine;
        Elf32_Word      e_version;
        Elf32_Addr      e_entry;
        Elf32_Off       e_phoff;
        Elf32_Off       e_shoff;
        Elf32_Word      e_flags;
        Elf32_Half      e_ehsize;
        Elf32_Half      e_phentsize;
        Elf32_Half      e_phnum;
        Elf32_Half      e_shentsize;
        Elf32_Half      e_shnum;
        Elf32_Half      e_shtrndx;
} Elf32_Ehdr;

typedef struct {
        unsigned char   e_ident[EI_NIDENT];
        Elf64_Half      e_type;
        Elf64_Half      e_machine;
        Elf64_Word      e_version;
        Elf64_Addr      e_entry;
        Elf64_Off       e_phoff;
        Elf64_Off       e_shoff;
        Elf64_Word      e_flags;
        Elf64_Half      e_ehsize;
        Elf64_Half      e_phentsize;
        Elf64_Half      e_phnum;
        Elf64_Half      e_shentsize;
        Elf64_Half      e_shnum;
        Elf64_Half      e_shtrndx;
} Elf64_Ehdr;
```

e_ident

The initial bytes mark the file as an object file and provide machine-independent data with which to decode and interpret the file's contents. Complete descriptions appear below in "ELF Identification" on page 903.

e_type

This member identifies the object file type.

Name	Value	Meaning
ET_NONE	**0**	No file type

Name	Value	Meaning
ET_REL	**1**	Relocatable file
ET_EXEC	**2**	Executable file
ET_DYN	**3**	Shared object file
ET_CORE	**4**	Core file
ET_LOOS	**0xfe00**	Operating system-specific
ET_HIOS	**0xfeff**	Operating system-specific
ET_LOPROC	**0xff00**	Processor-specific
ET_HIPROC	**0xffff**	Processor-specific

Although the core file contents are unspecified, type **ET_CORE** is reserved to mark the file. Values from **ET_LOOS** through **ET_HIOS** (inclusive) are reserved for operating system-specific semantics. Values from **ET_LOPROC** through **ET_HIPROC** (inclusive) are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them. Other values are reserved and will be assigned to new object file types as necessary.

e_machine

This member's value specifies the required architecture for an individual file.

Name	Value	Meaning
EM_NONE	**0**	No machine
EM_M32	**1**	AT&T WE 32100
EM_SPARC	**2**	SPARC
EM_386	**3**	Intel 80386
EM_68K	**4**	Motorola 68000
EM_88K	**5**	Motorola 88000
RESERVED	**6**	Reserved for future use
EM_860	**7**	Intel 80860
EM_MIPS	**8**	MIPS I Architecture
EM_S370	**9**	IBM System/370 Processor
EM_MIPS_RS3_LE	**10**	MIPS RS3000 Little-endian
RESERVED	**11-14**	Reserved for future use
EM_PARISC	**15**	Hewlett-Packard PA-RISC
RESERVED	**16**	Reserved for future use
EM_VPP500	**17**	Fujitsu VPP500
EM_SPARC32PLUS	**18**	Enhanced instruction set SPARC
EM_960	**19**	Intel 80960
EM_PPC	**20**	PowerPC
EM_PPC64	**21**	64-bit PowerPC
RESERVED	**22-35**	Reserved for future use
EM_V800	**36**	NEC V800
EM_FR20	**37**	Fujitsu FR20
EM_RH32	**38**	TRW RH-32

Name	Value	Meaning
EM_RCE	**39**	Motorola RCE
EM_ARM	**40**	Advanced RISC Machines ARM
EM_ALPHA	**41**	Digital Alpha
EM_SH	**42**	Hitachi SH
EM_SPARCV9	**43**	SPARC Version 9
EM_TRICORE	**44**	Siemens Tricore embedded processor
EM_ARC	**45**	Argonaut RISC Core, Argonaut Technologies Inc.
EM_H8_300	**46**	Hitachi H8/300
EM_H8_300H	**47**	Hitachi H8/300H
EM_H8S	**48**	Hitachi H8S
EM_H8_500	**49**	Hitachi H8/500
EM_IA_64	**50**	Itanium-based platform
EM_MIPS_X	**51**	Stanford MIPS-X
EM_COLDFIRE	**52**	Motorola ColdFire
EM_68HC12	**53**	Motorola M68HC12
EM_MMA	**54**	Fujitsu MMA Multimedia Accelerator
EM_PCP	**55**	Siemens PCP
EM_NCPU	**56**	Sony nCPU embedded RISC processor
EM_NDR1	**57**	Denso NDR1 microprocessor
EM_STARCORE	**58**	Motorola Star*Core processor
EM_ME16	**59**	Toyota ME16 processor
EM_ST100	**60**	STMicroelectronics ST100 processor
EM_TINYJ	**61**	Advanced Logic Corp. TinyJ embedded processor family
Reserved	**62-65**	Reserved for future use
EM_FX66	**66**	Siemens FX66 microcontroller
EM_ST9PLUS	**67**	STMicroelectronics ST9+ 8/16 bit microcontroller
EM_ST7	**68**	STMicroelectronics ST7 8-bit microcontroller
EM_68HC16	**69**	Motorola MC68HC16 Microcontroller
EM_68HC11	**70**	Motorola MC68HC11 Microcontroller
EM_68HC08	**71**	Motorola MC68HC08 Microcontroller
EM_68HC05	**72**	Motorola MC68HC05 Microcontroller
EM_SVX	**73**	Silicon Graphics SVx
EM_ST19	**74**	STMicroelectronics ST19 8-bit microcontroller
EM_VAX	**75**	Digital VAX
EM_CRIS	**76**	Axis Communications 32-bit embedded processor

Name	Value	Meaning
EM_JAVELIN	**77**	Infineon Technologies 32-bit embedded processor
EM_FIREPATH	**78**	Element 14 64-bit DSP Processor
EM_ZSP	**79**	LSI Logic 16-bit DSP Processor
EM_MMIX	**80**	Donald Knuth's educational 64-bit processor
EM_HUANY	**81**	Harvard University machine-independent object files
EM_PRISM	**82**	SiTera Prism

Other values are reserved and will be assigned to new machines as necessary. Processor-specific ELF names use the machine name to distinguish them. For example, the flags mentioned below use the prefix **EF_**; a flag named **WIDGET** for the **EM_XYZ** machine would be called **EF_XYZ_WIDGET**.

e_version
This member identifies the object file version.

Name	Value	Meaning
EV_NONE	**0**	Invalid version
EV_CURRENT	**1**	Current version

The value **1** signifies the original file format; extensions will create new versions with higher numbers. Although the value of **EV_CURRENT** is shown as **1** in the previous table, it will change as necessary to reflect the current version number.

e_entry
This member gives the virtual address to which the system first transfers control, thus starting the process. If the file has no associated entry point, this member holds zero.

e_phoff
This member holds the program header table's file offset in bytes. If the file has no program header table, this member holds zero.

e_shoff
This member holds the section header table's file offset in bytes. If the file has no section header table, this member holds zero.

e_flags
This member holds processor-specific flags associated with the file. Flag names take the form **EF**_*machine_flag*.

e_ehsize
This member holds the ELF header's size in bytes.

e_phentsize
This member holds the size in bytes of one entry in the file's program header table; all entries are the same size.

e_phnum
This member holds the number of entries in the program header table. Thus the product of **e_phentsize** and **e_phnum** gives the table's size in bytes. If a file has no program header table, **e_phnum** holds the value zero.

e_shentsize
This member holds a section header's size in bytes. A section header is one entry in the section header table; all entries are the same size.

e_shnum
This member holds the number of entries in the section header table. Thus the product of **e_shentsize** and **e_shnum** gives the section header table's size in bytes. If a file has no section header table, **e_shnum** holds the value zero.

e_shstrndx
This member holds the section header table index of the entry associated with the section name string table. If the file has no section name string table, this member holds the value **SHN_UNDEF**. See "Sections" on page 906 and "String Table" on page 917 for more information.

ELF Identification

As mentioned above, ELF provides an object file framework to support multiple processors, multiple data encodings, and multiple classes of machines. To support this object file family, the initial bytes of the file specify how to interpret the file, independent of the processor on which the inquiry is made and independent of the file's remaining contents.

The initial bytes of an ELF header (and an object file) correspond to the **e_ident** member.

e_ident[] Identification Indexes

Name	Value	Purpose
EI_MAG0	**0**	File identification
EI_MAG1	**1**	File identification
EI_MAG2	**2**	File identification
EI_MAG3	**3**	File identification
EI_CLASS	**4**	File class
EI_DATA	**5**	Data encoding
EI_VERSION	**6**	File version
EI_OSABI	**7**	Operating system/ABI identification
EI_ABIVERSION	**8**	ABI version
EI_PAD	**9**	Start of padding bytes
EI_NIDENT	**16**	Size of **e_ident[]**

These indexes access bytes that hold the following values.

EI_MAG0 to EI_MAG3
A file's first 4 bytes hold a magic number, identifying the file as an ELF object file.

Name	Value	Position
ELFMAG0	**0x7f**	**e_ident[EI_MAG0]**
ELFMAG1	**'E'**	**e_ident[EI_MAG1]**
ELFMAG2	**'L'**	**e_ident[EI_MAG2]**
ELFMAG3	**'F'**	**e_ident[EI_MAG3]**

EI_CLASS
The next byte, **e_ident[EI_CLASS]**, identifies the file's class, or capacity.

Name	Value	Meaning
ELFCLASSNONE	**0**	Invalid class
ELFCLASS32	**1**	32-bit objects
ELFCLASS64	**2**	64-bit objects

The file format is designed to be portable among machines of various sizes, without imposing the sizes of the largest machine on the smallest. The class of the file defines the basic types used by the data structures of the object file container itself. The data contained in object file sections may follow a different programming model. If so, the processor supplement describes the model used.

Class **ELFCLASS32** supports machines with 32-bit architectures. It uses the basic types defined in the table labeled 32-Bit Data Types.

Class **ELFCLASS64** supports machines with 64-bit architectures. It uses the basic types defined in the table labeled 64-Bit Data Types.

Other classes will be defined as necessary, with different basic types and sizes for object file data.

EI_DATA

Byte **e_ident[EI_DATA]** specifies the encoding of both the data structures used by object file container and data contained in object file sections.

The following encoding are currently defined.

Name	Value	Meaning
ELFDATANONE	**0**	Invalid data encoding
ELFDATA2LSB	**1**	See below
ELFDATA2MSB	**2**	See below

Other values are reserved and will be assigned to new encodings as necessary.

NOTE: Primarily for the convenience of code that looks at the ELF file at runtime, the ELF data structures are intended to have the same byte order as that of the running program.

EI_VERSION

Byte **e_ident[EI_VERSION]** specifies the ELF header version number. Currently, this value must be **EV_CURRENT**, as explained above for **e_version**.

EI_OSABI

Byte **e_ident[EI_OSABI]** identifies the operating system and ABI to which the object is targeted. Some fields in other ELF structures have flags and values that have operating system and/or ABI specific meanings; the interpretation of those fields is determined by the value of this byte. The value of this byte must be interpreted differently for each machine. That is, each value for the **e_machine** field determines a set of values for the **EI_OSABI** byte. Values are assigned by the ABI processor supplement for each machine. If the processor supplement does not specify a set of values, the value **0** shall be used and indicates *unspecified*.

EI_ABIVERSION

Byte **e_ident[EI_ABIVERSION]** identifies the version of the ABI to which the object is targeted. This field is used to distinguish among incompatible versions of an ABI. The interpretation of this version number is dependent on the ABI identified by the **EI_OSABI** field. If no values are specified for the **EI_OSABI** field by the processor supplement or no version values are specified for the ABI determined by a particular value of the **EI_OSABI** byte, the value **0** shall be used for the **EI_ABIVERSION** byte; it indicates *unspecified*.

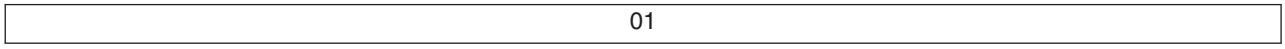
EI_PAD

This value marks the beginning of the unused bytes in **e_ident**. These bytes are reserved and set

to zero; programs that read object files should ignore them. The value of **EI_PAD** will change in the future if currently unused bytes are given meanings.

A file's data encoding specifies how to interpret the basic objects in a file. Class **ELFCLASS32** files use objects that occupy 1, 2, and 4 bytes. Class **ELFCLASS64** files use objects that occupy 1, 2, 4, and 8 bytes. Under the defined encodings, objects are represented as shown below.

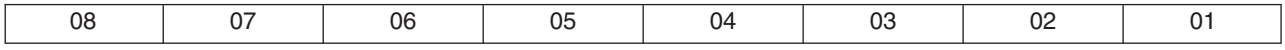
Encoding **ELFDATA2LSB** specifies 2's complement values, with the least significant byte occupying the lowest address.



0x01

0x0102

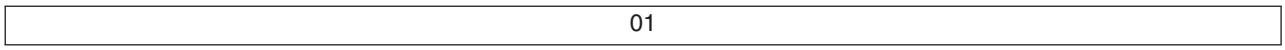
0x01020304



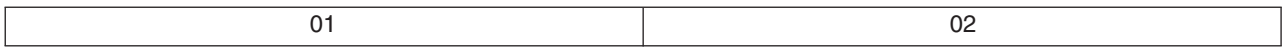
0x0102030405060708

Data Encoding ELFDATA2LSB, byte address zero on the left

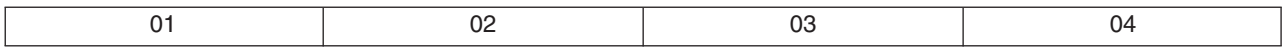
Encoding **ELFDATA2MSB** specifies 2's complement values, with the most significant byte occupying the lowest address.



0x01



0x0102



0x01020304

0x0102030405060708

Data Encoding ELFDATA2MSB, byte address zero on the left

Machine Information (Processor-Specific)

NOTE: This section requires processor-specific information. The ABI supplement for the desired processor describes the details.

Sections

An object file's section header table lets one locate all the file's sections. The section header table is an array of **Elf32_Shdr** or **Elf64_Shdr** structures as described below. A section header table index is a subscript into this array. The ELF header's **e_shoff** member gives the byte offset from the beginning of the file to the section header table. **e_shnum** tells how many entries the section header table contains. **e_shentsize** gives the size in bytes of each entry.

If the number of sections is greater than or equal to SHN_LORESERVE (0xff00), **e_shnum** has the value SHN_UNDEF (0) and the actual number of section header table entries is contained in the sh_size field of the section header at index 0 (otherwise, the **sh_size** member of the initial entry contains 0).

Some section header table indexes are reserved in contexts where index size is restricted, for example, the **st_shndx** member of a symbol table entry and the **e_shnum** and **e_shstrndx** members of the ELF header. In such contexts, the reserved values do not represent actual sections in the object file. Also in such contexts, an escape value indicates that the actual section index is to be found elsewhere, in a larger field.

Special Section Indexes

Name	Value
SHN_UNDEF	**0**
SHN_LORESERVE	**0xff00**
SHN_LOPROC	**0xff00**
SHN_HIPROC	**0xff1f**
SHN_LOOS	**0xff20**
SHN_HIOS	**0xff3f**
SHN_ABS	**0xfff1**
SHN_COMMON	**0xfff2**
SHN_HIRESERVE	**0xffff**

SHN_UNDEF
This value marks an undefined, missing, irrelevant, or otherwise meaningless section reference. For example, a symbol defined relative to section number **SHN_UNDEF** is an undefined symbol.

NOTE: Although index 0 is reserved as the undefined value, the section header table contains an entry for index 0. If the e_shnum member of the ELF header says a file has 6 entries in the section header table, they have the indexes 0 through 5. The contents of the initial entry are specified later in this section.

SHN_LORESERVE
This value specifies the lower bound of the range of reserved indexes.

SHN_LOPROC through SHN_HIPROC
Values in this inclusive range are reserved for processor-specific semantics.

SHN_LOOS through SHN_HIOS
Values in this inclusive range are reserved for operating system-specific semantics.

SHN_ABS
This value specifies absolute values for the corresponding reference. For example, symbols defined relative to section number equals **SHN_ABS** have absolute values and are not affected by relocation.

SHN_COMMON
Symbols defined relative to this section are common symbols, such as FORTRAN **COMMON** or unallocated C external variables.

SHN_HIRESERVE
This value specifies the upper bound of the range of reserved indexes. The system reserves indexes between **SHN_LORESERVE** and **SHN_HIRESERVE**, inclusive; the values do not reference the section header table. The section header table does not contain entries for the reserved indexes.

Sections contain all information in an object file except the ELF header, the program header table, and the section header table. Moreover, object files' sections satisfy several conditions.

- Every section in an object file has exactly one section header describing it. Section headers may exist that do not have a section.
- Each section occupies one contiguous (possibly empty) sequence of bytes within a file.
- Sections in a file may not overlap. No byte in a file resides in more than one section.
- An object file may have inactive space. The various headers and the sections might not cover every byte in an object file. The contents of the inactive data are unspecified.

A section header has the following structure.

```
typedef struct {
 Elf32_Word  sh_name;
 Elf32_Word  sh_type;
 Elf32_Word  sh_flags;
 Elf32_Addr  sh_addr;
 Elf32_Off  sh_offset;
 Elf32_Word  sh_size;
 Elf32_Word  sh_link;
 Elf32_Word  sh_info;
 Elf32_Word  sh_addralign;
 Elf32_Word  sh_entsize;
} Elf32_Shdr;

typedef struct {
 Elf64_Word  sh_name;
 Elf64_Word  sh_type;
 Elf64_Xword  sh_flags;
 Elf64_Addr  sh_addr;
 Elf64_Off  sh_offset;
 Elf64_Xword  sh_size;
 Elf64_Word  sh_link;
 Elf64_Word  sh_info;
 Elf64_Xword  sh_addralign;
 Elf64_Xword  sh_entsize;
} Elf64_Shdr;
```

Section Header

sh_name
This member specifies the name of the section. Its value is an index into the section header string table section [see "String Table" on page 917 below], giving the location of a null-terminated string.

sh_type
This member categorizes the section's contents and semantics. Section types and their descriptions appear below.

sh_flags
Sections support 1-bit flags that describe miscellaneous attributes. Flag definitions appear below.

sh_addr
If the section will appear in the memory image of a process, this member gives the address at which the section's first byte should reside. Otherwise, the member contains 0.

sh_offset
This member's value gives the byte offset from the beginning of the file to the first byte in the section. One section type, **SHT_NOBITS** described below, occupies no space in the file, and its **sh_offset** member locates the conceptual placement in the file.

sh_size
This member gives the section's size in bytes. Unless the section type is **SHT_NOBITS**, the section occupies **sh_size** bytes in the file. A section of type **SHT_NOBITS** may have a non-zero size, but it occupies no space in the file.

sh_link
This member holds a section header table index link, whose interpretation depends on the section type. A table below describes the values.

sh_info
This member holds extra information, whose interpretation depends on the section type. A table below describes the values. If the **sh_flags** field for this section header includes the attribute SHF_INFO_LINK, then this member represents a section header table index.

sh_addralign
Some sections have address alignment constraints. For example, if a section holds a doubleword, the system must ensure doubleword alignment for the entire section. The value of **sh_addr** must be congruent to 0, modulo the value of **sh_addralign**. Currently, only 0 and positive integral powers of two are allowed. Values 0 and 1 mean the section has no alignment constraints.

sh_entsize
Some sections hold a table of fixed-size entries, such as a symbol table. For such a section, this member gives the size in bytes of each entry. The member contains 0 if the section does not hold a table of fixed-size entries.

A section header's **sh_type** member specifies the section's semantics.

Section Types,sh_type

Name	Value
SHT_NULL	**0**
SHT_PROGBITS	**1**
SHT_SYMTAB	**2**
SHT_STRTAB	**3**
SHT_RELA	**4**
SHT_HASH	**5**
SHT_DYNAMIC	**6**
SHT_NOTE	**7**
SHT_NOBITS	**8**
SHT_REL	**9**

Name	Value
SHT_SHLIB	**10**
SHT_DYNSYM	**11**
SHT_INIT_ARRAY	**14**
SHT_FINI_ARRAY	**15**
SHT_PREINIT_ARRAY	**16**
SHT_LOOS	**0x60000000**
SHT_HIOS	**0x6fffffff**
SHT_LOPROC	**0x70000000**
SHT_HIPROC	**0x7fffffff**
SHT_LOUSER	**0x80000000**
SHT_HIUSER	**0xffffffff**

SHT_NULL
This value marks the section header as inactive; it does not have an associated section. Other members of the section header have undefined values.

SHT_PROGBITS
The section holds information defined by the program, whose format and meaning are determined solely by the program.

SHT_SYMTAB and SHT_DYNSYM
These sections hold a symbol table. Currently, an object file may have only one section of each type, but this restriction may be relaxed in the future. Typically, **SHT_SYMTAB** provides symbols for link editing, though it may also be used for dynamic linking. As a complete symbol table, it may contain many symbols unnecessary for dynamic linking. Consequently, an object file may also contain a **SHT_DYNSYM** section, which holds a minimal set of dynamic linking symbols, to save space. See the “Symbol Table” on page 920 for details.

SHT_STRTAB
The section holds a string table. An object file may have multiple string table sections. See the “String Table” on page 917 for details.

SHT_RELA
The section holds relocation entries with explicit addends, such as type **Elf32_Rela** for the 32-bit class of object files or type **Elf64_Rela** for the 64-bit class of object files. An object file may have multiple relocation sections. See “Relocation” on page 918 for details.

SHT_HASH
The section holds a symbol hash table. Currently, an object file may have only one hash table, but this restriction may be relaxed in the future. See the “Hash Table” on page 940 in Chapter 32 for details.

SHT_DYNAMIC
The section holds information for dynamic linking. Currently, an object file may have only one dynamic section, but this restriction may be relaxed in the future. See “Dynamic Section” on page 933 in chapter 32 for details.

SHT_NOTE
The section holds information that marks the file in some way. See “Note Section” on page 930 in Chapter 32 for details.

SHT_NOBITS
A section of this type occupies no space in the file but otherwise resembles **SHT_PROGBITS**. Although this section contains no bytes, the **sh_offset** member contains the conceptual file offset.

SHT_REL
The section holds relocation entries without explicit addends, such as type **Elf32_Rel** for the 32-bit class of object files or type **Elf64_Rel** for the 64-bit class of object files. An object file may have multiple relocation sections. See “Relocation” on page 918 for details.

SHT_SHLIB
This section type is reserved but has unspecified semantics.

SHT_INIT_ARRAY
This section contains an array of pointers to initialization functions, as described in “Initialization and Termination Functions” on page 941 in Chapter 32. Each pointer in the array is taken as a parameterless procedure with a void return.

SHT_FINI_ARRAY
This section contains an array of pointers to termination functions, as described in “Initialization and Termination Functions” on page 941 in chapter 32. Each pointer in the array is taken as a parameterless procedure with a void return.

SHT_PREINIT_ARRAY
This section contains an array of pointers to functions that are invoked before all other initialization functions, as described in “Initialization and Termination Functions” on page 941 in Chapter 32. Each pointer in the array is taken as a parameterless procedure with a void return.

SHT_LOOS through SHT_HIOS
Values in this inclusive range are reserved for operating system-specific semantics.

SHT_LOPROC through SHT_HIPROC
Values in this inclusive range are reserved for processor-specific semantics.

SHT_LOUSER
This value specifies the lower bound of the range of indexes reserved for application programs.

SHT_HIUSER
This value specifies the upper bound of the range of indexes reserved for application programs. Section types between **SHT_LOUSER** and **SHT_HIUSER** may be used by the application, without conflicting with current or future system-defined section types.

Other section type values are reserved. As mentioned before, the section header for index 0 (**SHN_UNDEF**) exists, even though the index marks undefined section references. This entry holds the following.

Section Header Table Entry:Index 0

Name	Value	Note
sh_name	**0**	No name
sh_type	**SHT_NULL**	Inactive
sh_flags	**0**	No flags
sh_addr	**0**	No address
sh_offset	**0**	No offset
sh_size	**Unspecified**	If non-zero, the actual number of section header entries
sh_link	**Unspecified**	If non-zero, the index of the section header string table section
sh_info	**0**	No auxiliary information
sh_addralign	**0**	No alignment
sh_entsize	**0**	No entries

A section header's **sh_flags** member holds 1-bit flags that describe the section's attributes. Defined values appear in the following table; other values are reserved.

Section Attribute Flags

Name	Value
SHF_WRITE	**0x1**
SHF_ALLOC	**0x2**
SHF_EXECINSTR	**0x4**
SHF_MERGE	**0x10**
SHF_STRINGS	**0x20**
SHF_INFO_LINK	**0x40**
SHF_LINK_ORDER	**0x80**
SHF_OS_NONCONFORMING	**0x100**
SHF_GROUP	**0x200**
SHF_MASKOS	**0x0ff00000**
SHF_MASKPROC	**0xf0000000**

If a flag bit is set in **sh_flags**, the attribute is on for the section. Otherwise, the attribute is off or does not apply. Undefined attributes are set to zero.

SHF_WRITE
The section contains data that should be writable during process execution.

SHF_ALLOC
The section occupies memory during process execution. Some control sections do not reside in the memory image of an object file; this attribute is off for those sections.

SHF_EXECINSTR
The section contains executable machine instructions.

SHF_MERGE
The data in the section may be merged to eliminate duplication. Unless the **SHF_STRINGS** flag is also set, the data elements in the section are of a uniform size. The size of each element is specified in the section header's **sh_entsize** field. If the **SHF_STRINGS** flag is also set, the data elements consist of null-terminated character strings. The size of each character is specified in the section header's **sh_entsize** field.

Each element in the section is compared against other elements in sections with the same name, type and flags. Elements that would have identical values at program run-time may be merged. Relocations referencing elements of such sections must be resolved to the merged locations of the referenced values. Note that any relocatable values, including values that would result in run-time relocations, must be analyzed to determine whether the run-time values would actually be identical. An ABI-conforming object file may not depend on specific elements being merged, and an ABI-conforming link editor may choose not to merge specific elements.

SHF_STRINGS
The data elements in the section consist of null-terminated character strings. The size of each character is specified in the section header's **sh_entsize** field.

SHF_INFO_LINK
The **sh_info** field of this section header holds a section header table index.

SHF_LINK_ORDER
This flag adds special ordering requirements for link editors. The requirements apply if the **sh_link** field of this section's header references another section (the linked-to section). If this section is

combined with other sections in the output file, it must appear in the same relative order with respect to those sections, as the linked-to section appears with respect to sections the linked-to section is combined with.

NOTE: A typical use of this flag is to build a table that references text or data sections in address order.

SHF_OS_NONCONFORMING
This section requires special operating system specific processing (beyond the standard linking rules ("Rules for Linking Unrecognized Sections") to avoid incorrect behavior. If this section has either an **sh_type** value or contains **sh_flags** bits in the OS-specific ranges for those fields, and a link editor processing this section does not recognize those values, then the link editor should reject the object file containing this section with an error.

SHF_MASKOS
All bits included in this mask are reserved for operating system-specific semantics.

SHF_MASKPROC
All bits included in this mask are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.

Two members in the section header, **sh_link** and **sh_info**, hold special information, depending on section type.

sh_link and sh_info Interpretation

sh_type	sh_link	sh_info
SHT_DYNAMIC	The section header index of the string table used by entries in the section.	**0**
SHT_HASH	The section header index of the symbol table to which the hash table applies.	**0**
SHT_REL **SHT_RELA**	The section header index of the associated symbol table.	The section header index of the section to which the relocation applies.
SHT_SYMTAB **SHT_DYNSYM**	The section header index of the associated string table.	One greater than the symbol table index of the last local symbol (binding **STB_LOCAL**).
SHT_GROUP	The section header index of the associated symbol table.	The symbol table index of an entry in the associated symbol table. The name of the specified symbol table entry provides a signature for the section group.
SHT_SYMTAB_SHNDX	The section header index of the associated symbol table section.	0

Rules for Linking Unrecognized Sections

If a link editor encounters sections whose headers contain operating system specific values it does not recognize in the **sh_type** or **sh_flags** fields, the link editor should combine those sections as described below.

If the section's **sh_flags** bits include the attribute **SHF_OS_NONCONFORMING**, then the section requires special knowledge to be correctly processed, and the link editor should reject the object containing the section with an error.

Unrecognized sections that do not have the **SHF_OS_NONCONFORMING** attribute, are combined in a two-phase process. As the link editor combines sections using this process, it must honor the alignment constraints of the input sections (asserted by the **sh_addralign** field), padding between sections with zero bytes, if necessary, and producing a combination with the maximum alignment constraint of its component input sections.

1. In the first phase, input sections that match in name, type and attribute flags should be concatenated into single sections. The concatenation order should satisfy the requirements of any known input section attributes (e.g, **SHF_MERGE** and **SHF_LINK_ORDER**). When not otherwise constrained, sections should be emitted in input order.
2. In the second phase, sections should be assigned to segments or other units based on their attribute flags. Sections of each particular unrecognized type should be assigned to the same unit unless prevented by incompatible flags, and within a unit, sections of the same unrecognized type should be placed together if possible.

Non operating system specific processing (e.g. relocation) should be applied to unrecognized section types. An output section header table, if present, should contain entries for unknown sections. Any unrecognized section attribute flags should be removed.

NOTE: It is recommended that link editors follow the same two-phase ordering approach described above when linking sections of known types. Padding between such sections may have values different from zero, where appropriate.

Section Groups

Some sections occur in interrelated groups. For example, an out-of-line definition of an inline function might require, in addition to the section containing its executable instructions, a read-only data section containing literals referenced, one or more debugging information sections and other informational sections. Furthermore, there may be internal references among these sections that would not make sense if one of the sections were removed or replaced by a duplicate from another object. Therefore, such groups must be included or omitted from the linked object as a unit.

A section of type SHT_GROUP defines such a grouping of sections. The name of a symbol from one of the containing object's symbol tables provides a signature for the section group. The section header of the SHT_GROUP section specifies the identifying symbol entry, as described above: the sh_link member contains the section header index of the symbol table section that contains the entry. The sh_info member contains the symbol table index of the identifying entry. The sh_flags member of the section header contains 0. The name of the section (sh_name) is not specified.

The section data of a SHT_GROUP section is an array of Elf32_Word entries. The first entry is a flag word. The remaining entries are a sequence of section header indices.

The following flags are currently defined:

Section Group Flags

Name	Value
GRP_COMDAT	0x1

GRP_COMDAT

This is a COMDAT group. It may duplicate another COMDAT group in another object file, where duplication is defined as having the same group signature. In such cases, only one of the duplicate groups may be retained by the linker, and the members of the remaining groups must be discarded.

The section header indices in the SHT_GROUP section identify the sections that make up the group. Each such section must have the SHF_GROUP flag set in its sh_flags section header member. If the linker decides to remove the section group, it must remove all members of the group.

Note: This requirement is not intended to imply that special case behavior like removing debugging information requires removing the sections to which that information refers, even if they are part of the same group.

To facilitate removing a group without leaving dangling references and with only minimal processing of the symbol table, the following rules must be followed:

- References to the sections comprising a group from sections outside of the group must be made via symbol table entries with STB_GLOBAL or STB_WEAK binding and section index SHN_UNDEF. If there is a definition of the same symbol in the object containing the references, it must have a separate symbol table entry from the references. Sections outside of the group may not reference symbols with STB_LOCAL binding for addresses contained in the group's sections, including symbols with type STT_SECTION.
- There may not be non-symbol references to the sections comprising a group from outside the group, for example, use of a group member's section header index in an sh_link or sh_info member.
- A symbol table entry that is defined relative to one of the group's sections and that is contained in a symbol table section that is not part of the group, must be removed if the group members are discarded.

Special Sections

Various sections hold program and control information.

The following table shows sections that are used by the system and have the indicated types and attributes.

Special Sections

Name	Type	Attributes
.bss	**SHT_NOBITS**	**SHF_ALLOC+SHF_WRITE**
.comment	**SHT_PROGBITS**	none
.data	**SHT_PROGBITS**	**SHF_ALLOC+SHF_WRITE**
.data1	**SHT_PROGBITS**	**SHF_ALLOC+SHF_WRITE**
.debug	**SHT_PROGBITS**	none
.dynamic	**SHT_DYNAMIC**	see below
.dynstr	**SHT_STRTAB**	**SHF_ALLOC**
.dynsym	**SHT_DYNSYM**	**SHF_ALLOC**
.fini	**SHT_PROGBITS**	**SHF_ALLOC+SHF_EXECINSTR**
.fini_array	**SHT_FINI_ARRAY**	**SHF_ALLOC+SHF_WRITE**
.got	**SHT_PROGBITS**	see below
.hash	**SHT_HASH**	**SHF_ALLOC**
.init	**SHT_PROGBITS**	**SHF_ALLOC+SHF_EXECINSTR**
.init_array	**SHT_INIT_ARRAY**	**SHF_ALLOC+SHF_WRITE**
.interp	**SHT_PROGBITS**	see below
.line	**SHT_PROGBITS**	none
.note	**SHT_NOTE**	none
.plt	**SHT_PROGBITS**	see below

Name	Type	Attributes
.preinit_array	**SHT_PREINIT_ARRAY**	**SHF_ALLOC+SHF_WRITE**
.rel*name*	**SHT_REL**	see below
.rela*name*	**SHT_RELA**	see below
.rodata	**SHT_PROGBITS**	**SHF_ALLOC**
.rodata1	**SHT_PROGBITS**	**SHF_ALLOC**
.shstrtab	**SHT_STRTAB**	none
.strtab	**SHT_STRTAB**	see below
.symtab	**SHT_SYMTAB**	see below
.text	**SHT_PROGBITS**	**SHF_ALLOC+SHF_EXECINSTR**

.bss This section holds uninitialized data that contribute to the program's memory image. By definition, the system initializes the data with zeros when the program begins to run. The section occupies no file space, as indicated by the section type, **SHT_NOBITS**.

.comment
This section holds version control information.

.data and .data1
These sections hold initialized data that contribute to the program's memory image.

.debug
This section holds information for symbolic debugging. The contents are unspecified. All section names with the prefix **.debug** are reserved for future use in the ABI.

.dynamic
This section holds dynamic linking information. The section's attributes will include the **SHF_ALLOC** bit. Whether the **SHF_WRITE** bit is set is processor specific. See chapter 32 for more information.

.dynstr
This section holds strings needed for dynamic linking, most commonly the strings that represent the names associated with symbol table entries. See chapter 32 for more information.

.dynsym
This section holds the dynamic linking symbol table, as described in "Symbol Table" on page 920. See Chapter 32 for more information.

.fini This section holds executable instructions that contribute to the process termination code. That is, when a program exits normally, the system arranges to execute the code in this section.

.fini_array
This section holds an array of function pointers that contributes to a single termination array for the executable or shared object containing the section.

.got This section holds the global offset table. See Coding Examples in chapter 32, Special Sections in chapter 32, and Global Offset Table in chapter 32 of the processor supplement for more information.

.hash This section holds a symbol hash table. See "Hash Table" on page 940 in chapter 32 for more information.

.init This section holds executable instructions that contribute to the process initialization code. When a program starts to run, the system arranges to execute the code in this section before calling the main program entry point (called **main** for C programs).

.init_array
This section holds an array of function pointers that contributes to a single initialization array for the executable or shared object containing the section.

.interp
This section holds the path name of a program interpreter. If the file has a loadable segment that includes relocation, the sections' attributes will include the **SHF_ALLOC** bit; otherwise, that bit will be off. See chapter 32 for more information.

.line This section holds line number information for symbolic debugging, which describes the correspondence between the source program and the machine code. The contents are unspecified.

.note This section holds information in the format that Chapter 32 describes. See "Note Section" on page 930.

.plt This section holds the procedure linkage table. See ″Special Sections″ in Chapter 32 and the ″Procedure Linkage Table″ in chapter 32 of the processor supplement for more information.

.preinit_array
This section holds an array of function pointers that contributes to a single pre-initialization array for the executable or shared object containing the section.

.rel*name* **and .rela***name*
These sections hold relocation information, as described in "Relocation" on page 918. If the file has a loadable segment that includes relocation, the sections' attributes will include the **SHF_ALLOC** bit; otherwise, that bit will be off. Conventionally, *name* is supplied by the section to which the relocations apply. Thus a relocation section for **.text** normally would have the name **.rel.text** or **.rela.text**.

.rodata and .rodata1
These sections hold read-only data that typically contribute to a non-writable segment in the process image. See "Program Header" on page 926 in Chapter 32 for more information.

.shstrtab
This section holds section names.

.strtab
This section holds strings, most commonly the strings that represent the names associated with symbol table entries. If the file has a loadable segment that includes the symbol string table, the section's attributes will include the **SHF_ALLOC** bit; otherwise, that bit will be off.

.symtab
This section holds a symbol table, as "Symbol Table" on page 920 in this chapter describes. If the file has a loadable segment that includes the symbol table, the section's attributes will include the **SHF_ALLOC** bit; otherwise, that bit will be off.

.text This section holds the text, or executable instructions, of a program.

Section names with a dot (**.**) prefix are reserved for the system, although applications may use these sections if their existing meanings are satisfactory. Applications may use names without the prefix to avoid conflicts with system sections. The object file format lets one define sections not shown in the previous list. An object file may have more than one section with the same name.

Section names reserved for a processor architecture are formed by placing an abbreviation of the architecture name ahead of the section name. The name should be taken from the architecture names used for **e_machine**. For instance **.**FOO**.psect** is the **psect** section defined by the FOO architecture.

Existing extensions are called by their historical names.

Table 11. **Pre-existing Extensions**

.sdata	**.tdesc**
.sbss	**.lit4**
.lit8	**.reginfo**
.gptab	**.liblist**
.conflict	

NOTE: For information on processor-specific sections, see the ABI supplement for the desired processor.

String Table

String table sections hold null-terminated character sequences, commonly called strings. The object file uses these strings to represent symbol and section names. One references a string as an index into the string table section. The first byte, which is index zero, is defined to hold a null character. Likewise, a string table's last byte is defined to hold a null character, ensuring null termination for all strings. A string whose index is zero specifies either no name or a null name, depending on the context. An empty string table section is permitted; its section header's `sh_size` member would contain zero. Non-zero indexes are invalid for an empty string table.

A section header's `sh_name` member holds an index into the section header string table section, as designated by the `e_shstrndx` member of the ELF header. The following tables show a string table with 25 bytes and the strings associated with various indexes.

Index	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	\0	n	a	m	e	.	\0	V	a	r
10	i	a	b	l	e	\0	a	b	l	e
20	\0	\0	x	x	\0					

String Indexes

Index	String
0	*none*
1	name.
7	Variable
11	able
16	able
24	*null string*

As the example shows, a string table index may refer to any byte in the section. A string may appear more than once; references to substrings may exist; and a single string may be referenced multiple times. Unreferenced strings also are allowed.

System V Application Binary Interface

This topic describes the executable and linking format (ELF) of the object files produced by the C and C++ compilation system.

There are three main types of object files.

- A relocatable file holds code and data suitable for linking with other object files to create an executable or a shared object file.
- An executable file holds a program suitable for execution; the file specifies how **exec** creates a program's process image.
- A shared object file holds code and data suitable for linking in two contexts. First, the link editor processes the shared object file with other relocatable and shared object files to create another object file. Second, the dynamic linker combines it with an executable file and other shared objects to create a process image.

Programs manipulate object files with the functions contained in the ELF access library, **libelf**.

See **Intro**(elf) for more information.

> NOTE: Further information is available in the System V Application Binary Interface and processor specific supplements. The processor supplements define a naming convention for ELF constants that have processor ranges specified. Names such as DT_ and PT_ for processor specific extensions incorporate the name of the processor.

Relocation

Relocation is the process of connecting symbolic references with symbolic definitions. For example, when a program calls a function, the associated call instruction must transfer control to the proper destination address at execution. Relocatable files must have relocation entries which are necessary because they contain information that describes how to modify their section contents, thus allowing executable and shared object files to hold the right information for a process's program image.

Relocation Entries

```
typedef struct {
   Elf32_Addr   r_offset;
   Elf32_Word   r_info;
} Elf32_Rel;
typedef struct {
   Elf32_Addr   r_offset;
   Elf32_Word   r_info;
   Elf32_Sword   r_addend;
} Elf32_Rela;
typedef struct {
   Elf64_Addr   r_offset;
   Elf64_Xword   r_info;
} Elf64_Rel;
typedef struct {
   Elf64_Addr   r_offset;
   Elf64_Xword   r_info;
   Elf64_Sxword   r_addend;
} Elf64_Rela;
```

r_offset
: This member gives the location at which to apply the relocation action. For a relocatable file, the value is the byte offset from the beginning of the section to the storage unit affected by the relocation. For an executable file or a shared object, the value is the virtual address of the storage unit affected by the relocation.

r_info This member gives both the symbol table index with respect to which the relocation must be

made, and the type of relocation to apply. For example, a call instruction's relocation entry would hold the symbol table index of the function being called. If the index is **STN_UNDEF**, the undefined symbol index, the relocation uses 0 as the symbol value. Relocation types are processor-specific; descriptions of their behavior appear in the processor supplement. When the text below refers to a relocation entry's relocation type or symbol table index, it means the result of applying **ELF32_R_TYPE** (or **ELF64_R_TYPE**) or **ELF32_R_SYM** (or **ELF64_R_SYM**), respectively, to the entry's **r_info** member.

```
#define ELF32_R_SYM(i)  ((i)>>8)
#define ELF32_R_TYPE(i)   ((unsigned char)(i))
#define ELF32_R_INFO(s,t) (((s)<<8)+(unsigned char)(t))

#define ELF64_R_SYM(i)    ((i)>>32)
#define ELF64_R_TYPE(i)   ((i)&0xffffffffL)
#define ELF64_R_INFO(s,t) (((s)<<32)+((t)&0xffffffffL))
```

r_addend

This member specifies a constant addend used to compute the value to be stored into the relocatable field.

As specified previously, only **Elf32_Rela** and **Elf64_Rela** entries contain an explicit addend. Entries of type **Elf32_Rel** and **Elf64_Rel** store an implicit addend in the location to be modified. Depending on the processor architecture, one form or the other might be necessary or more convenient. Consequently, an implementation for a particular machine may use one form exclusively or either form depending on context.

A relocation section references two other sections: a symbol table and a section to modify. The section header's **sh_info** and **sh_link** members, described in "Sections" on page 906 above, specify these relationships. Relocation entries for different object files have slightly different interpretations for the **r_offset** member.

- In relocatable files, **r_offset** holds a section offset. The relocation section itself describes how to modify another section in the file; relocation offsets designate a storage unit within the second section.
- In executable and shared object files, **r_offset** holds a virtual address. To make these files' relocation entries more useful for the dynamic linker, the section offset (file interpretation) gives way to a virtual address (memory interpretation).

Although the interpretation of **r_offset** changes for different object files to allow efficient access by the relevant programs, the relocation types' meanings stay the same.

The typical application of an ELF relocation is to determine the referenced symbol value, extract the addend (either from the field to be relocated or from the addend field contained in the relocation record, as appropriate for the type of relocation record), apply the expression implied by the relocation type to the symbol and addend, extract the desired part of the expression result, and place it in the field to be relocated.

If multiple *consecutive* relocation records are applied to the same relocation location (**r_offset**), they are *composed* instead of being applied independently, as described above. By *consecutive*, we mean that the relocation records are contiguous within a single relocation section. By *composed*, we mean that the standard application described above is modified as follows:

- In all but the last relocation operation of a composed sequence, the result of the relocation expression is retained, rather than having part extracted and placed in the relocated field. The result is retained at full pointer precision of the applicable ABI processor supplement.
- In all but the first relocation operation of a composed sequence, the addend used is the retained result of the previous relocation operation, rather than that implied by the relocation type.

Note that a consequence of the above rules is that the location specified by a relocation type is relevant for the first element of a composed sequence (and then only for relocation records that do not contain an explicit addend field) and for the last element, where the location determines where the relocated value will be placed. For all other relocation operands in a composed sequence, the location specified is ignored.

An ABI processor supplement may specify individual relocation types that always stop a composition sequence, or always start a new one.

Relocation Types (Processor-Specific)

NOTE: This section requires processor-specific information. The ABI supplement for the desired processor describes the details.

Symbol Table

An object file's symbol table holds information needed to locate and relocate a program's symbolic definitions and references. A symbol table index is a subscript into this array. Index 0 both designates the first entry in the table and serves as the undefined symbol index. The contents of the initial entry are specified later in this section.

Name	Value
STN_UNDEF	**0**

A symbol table entry has the following format.

```
typedef struct {
 Elf32_Word  st_name;
 Elf32_Addr  st_value;
 Elf32_Word  st_size;
 unsigned char  st_info;
 unsigned char  st_other;
 Elf32_Half  st_shndx;
} Elf32_Sym;

typedef struct {
 Elf64_Word  st_name;
 unsigned char  st_info;
 unsigned char  st_other;
 Elf64_Half  st_shndx;
 Elf64_Addr  st_value;
 Elf64_Xword  st_size;
} Elf64_Sym;
```

Symbol Table Entry

st_name

This member holds an index into the object file's symbol string table, which holds the character representations of the symbol names. If the value is non-zero, it represents a string table index that gives the symbol name. Otherwise, the symbol table entry has no name.

NOTE: External C symbols have the same names in C and object files' symbol tables.

st_value

This member gives the value of the associated symbol. Depending on the context, this may be an absolute value, an address, and so on; details appear below.

st_size

Many symbols have associated sizes. For example, a data object's size is the number of bytes contained in the object. This member holds 0 if the symbol has no size or an unknown size.

st_info

This member specifies the symbol's type and binding attributes. A list of the values and meanings appears below. The following code shows how to manipulate the values for both 32 and 64-bit objects.

```
#define ELF32_ST_BIND(i)   ((i)>>4)
#define ELF32_ST_TYPE(i)   ((i)&0xf)
#define ELF32_ST_INFO(b,t) (((b)<<4)+((t)&0xf))

#define ELF64_ST_BIND(i)   ((i)>>4)
#define ELF64_ST_TYPE(i)   ((i)&0xf)
#define ELF64_ST_INFO(b,t) (((b)<<4)+((t)&0xf))
```

st_other

This member currently specifies a symbol's visibility. A list of the values and meanings appears below. The following code shows how to manipulate the values for both 32 and 64-bit objects. Other bits contain 0 and have no defined meaning.

```
#define ELF32_ST_VISIBILITY(o) ((o)&0x3)
#define ELF32_ST_OTHER(v)      ((v)&0x3)

#define ELF64_ST_VISIBILITY(o) ((o)&0x3)
#define ELF64_ST_OTHER(v)      ((v)&0x3)
```

st_shndx

Every symbol table entry is *defined* in relation to some section. This member holds the relevant section header table index. As the **sh_link** and **sh_info** interpretation table and the related text describe, some section indexes indicate special meanings.

If this member contains SHN_XINDEX, then the actual section header index is too large to fit in this field. The actual value is contained in the associated section of type SHT_SYMTAB_SHNDX.

A symbol's binding determines the linkage visibility and behavior.

Symbol Binding

Name	Value
STB_LOCAL	**0**
STB_GLOBAL	**1**
STB_WEAK	**2**
STB_LOOS	**10**
STB_HIOS	**12**
STB_LOPROC	**13**
STB_HIPROC	**15**

STB_LOCAL

Local symbols are not visible outside the object file containing their definition. Local symbols of the same name may exist in multiple files without interfering with each other.

STB_GLOBAL

Global symbols are visible to all object files being combined. One file's definition of a global symbol will satisfy another file's undefined reference to the same global symbol.

STB_WEAK

Weak symbols resemble global symbols, but their definitions have lower precedence.

STB_LOOS through STB_HIOS

Values in this inclusive range are reserved for operating system-specific semantics.

STB_LOPROC through STB_HIPROC
Values in this inclusive range are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.

Global and weak symbols differ in two major ways.

- When the link editor combines several relocatable object files, it does not allow multiple definitions of **STB_GLOBAL** symbols with the same name. On the other hand, if a defined global symbol exists, the appearance of a weak symbol with the same name will not cause an error. The link editor honors the global definition and ignores the weak ones. Similarly, if a common symbol exists (that is, a symbol whose **st_shndx** field holds **SHN_COMMON**), the appearance of a weak symbol with the same name will not cause an error. The link editor honors the common definition and ignores the weak ones.
- When the link editor searches archive libraries [see Archive File in Chapter 7], it extracts archive members that contain definitions of undefined global symbols. The member's definition may be either a global or a weak symbol. The link editor does not extract archive members to resolve undefined weak symbols. Unresolved weak symbols have a zero value.

NOTE: The behavior of weak symbols in areas not specified by this document is implementation defined. Weak symbols are intended primarily for use in system software. Their use in application programs is discouraged.

In each symbol table, all symbols with **STB_LOCAL** binding precede the weak and global symbols. As "Sections" on page 906, above describes, a symbol table section's **sh_info** section header member holds the symbol table index for the first non-local symbol.

A symbol's type provides a general classification for the associated entity.

Symbol Types

Name	Value
STT_NOTYPE	**0**
STT_OBJECT	**1**
STT_FUNC	**2**
STT_SECTION	**3**
STT_FILE	**4**
STT_COMMON	**5**
STT_LOOS	**10**
STT_HIOS	**12**
STT_LOPROC	**13**
STT_HIPROC	**15**

STT_NOTYPE
The symbol's type is not specified.

STT_OBJECT
The symbol is associated with a data object, such as a variable, an array, and so on.

STT_FUNC
The symbol is associated with a function or other executable code.

STT_SECTION
The symbol is associated with a section. Symbol table entries of this type exist primarily for relocation and normally have **STB_LOCAL** binding.

STT_FILE
Conventionally, the symbol's name gives the name of the source file associated with the object file. A file symbol has **STB_LOCAL** binding, its section index is **SHN_ABS**, and it precedes the other **STB_LOCAL** symbols for the file, if it is present.

STT_COMMON
The symbol labels an uninitialized common block. See below for details.

STT_LOOS through STT_HIOS
Values in this inclusive range are reserved for operating system-specific semantics.

STT_LOPROC through STT_HIPROC
Values in this inclusive range are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.

Function symbols (those with type **STT_FUNC**) in shared object files have special significance. When another object file references a function from a shared object, the link editor automatically creates a procedure linkage table entry for the referenced symbol. Shared object symbols with types other than **STT_FUNC** will not be referenced automatically through the procedure linkage table.

Symbols with type **STT_COMMON** label uninitialized common blocks. In relocatable objects, these symbols are not allocated and must have the special section index **SHN_COMMON** (see below). In shared objects and executables these symbols must be allocated to some section in the defining object.

In relocatable objects, symbols with type **STT_COMMON** are treated just as other symbols with index **SHN_COMMON**. If the link-editor allocates space for the **SHN_COMMON** symbol in an output section of the object it is producing, it must preserve the type of the output symbol as **STT_COMMON**.

When the dynamic linker encounters a reference to a symbol that resolves to a definition of type **STT_COMMON**, it may (but is not required to) change its symbol resolution rules as follows: instead of binding the reference to the first symbol found with the given name, the dynamic linker searches for the first symbol with that name with type other than **STT_COMMON**. If no such symbol is found, it looks for the **STT_COMMON** definition of that name that has the largest size.

A symbol's visibility, although it may be specified in a relocatable object, defines how that symbol may be accessed once it has become part of an executable or shared object.

Symbol Visibility

Name	Value
STV_DEFAULT	**0**
STV_INTERNAL	**1**
STV_HIDDEN	**2**
STV_PROTECTED	**3**

STV_DEFAULT
The visibility of symbols with the **STV_DEFAULT** attribute is as specified by the symbol's binding type. That is, global and weak symbols are visible outside of their defining *component* (executable file or shared object). Local symbols are *hidden*, as described below. Global and weak symbols are also *preemptable*, that is, they may by preempted by definitions of the same name in another component.

NOTE: An implementation may restrict the set of global and weak symbols that are externally visible.

STV_PROTECTED
A symbol defined in the current component is *protected* if it is visible in other components but not

preemptable, meaning that any reference to such a symbol from within the defining component must be resolved to the definition in that component, even if there is a definition in another component that would preempt by the default rules. A symbol with **STB_LOCAL** binding may not have **STV_PROTECTED** visibility.

STV_HIDDEN
A symbol defined in the current component is *hidden* if its name is not visible to other components. Such a symbol is necessarily protected. This attribute may be used to control the external interface of a component. Note that an object named by such a symbol may still be referenced from another component if its address is passed outside.

A hidden symbol contained in a relocatable object must be either removed or converted to **STB_LOCAL** binding by the link-editor when the relocatable object is included in an executable file or shared object.

STV_INTERNAL
The meaning of this visibility attribute may be defined by processor supplements to further constrain hidden symbols. A processor supplement's definition should be such that generic tools can safely treat internal symbols as hidden.

An internal symbol contained in a relocatable object must be either removed or converted to **STB_LOCAL** binding by the link-editor when the relocatable object is included in an executable file or shared object.

None of the visibility attributes affects resolution of symbols within an executable or shared object during link-editing — such resolution is controlled by the binding type. Once the link-editor has chosen its resolution, these attributes impose two requirements, both based on the fact that references in the code being linked may have been optimized to take advantage of the attributes.

- First, all of the non-default visibility attributes, when applied to a symbol reference, imply that a definition to satisfy that reference must be provided within the current executable or shared object. If such a symbol reference has no definition within the component being linked, then the reference must have **STB_WEAK** binding and is resolved to zero.
- Second, if any reference to or definition of a name is a symbol with a non-default visibility attribute, the visibility attribute must be propagated to the resolving symbol in the linked object. If different visibility attributes are specified for distinct references to or definitions of a symbol, the most constraining visibility attribute must be propagated to the resolving symbol in the linked object. The attributes, ordered from least to most constraining, are: **STV_PROTECTED**, **STV_HIDDEN** and **STV_INTERNAL**.

If a symbol's value refers to a specific location within a section, its section index member, **st_shndx**, holds an index into the section header table. As the section moves during relocation, the symbol's value changes as well, and references to the symbol continue to point to the same location in the program. Some special section index values give other semantics.

SHN_ABS
The symbol has an absolute value that will not change because of relocation.

SHN_COMMON
The symbol labels a common block that has not yet been allocated. The symbol's value gives alignment constraints, similar to a section's **sh_addralign** member. The link editor will allocate the storage for the symbol at an address that is a multiple of **st_value**. The symbol's size tells how many bytes are required. Symbols with section index **SHN_COMMON** may appear only in relocatable objects.

SHN_UNDEF
This section table index means the symbol is undefined. When the link editor combines this object file with another that defines the indicated symbol, this file's references to the symbol will be linked to the actual definition.

The symbol table entry for index 0 (**STN_UNDEF**) is reserved; it holds the following.

Symbol Entry:Index 0

Name	Value	Note
st_name	**0**	No name
st_value	**0**	Zero value
st_size	**0**	No size
st_info	**0**	No type, local binding
st_other	**0**	Default visibility
st_shndx	**SHN_UNDEF**	No section

Symbol Values

Symbol table entries for different object file types have slightly different interpretations for the **st_value** member.

- In relocatable files, **st_value** holds alignment constraints for a symbol whose section index is **SHN_COMMON**.
- In relocatable files, **st_value** holds a section offset for a defined symbol. **st_value** is an offset from the beginning of the section that **st_shndx** identifies.
- In executable and shared object files, **st_value** holds a virtual address. To make these files' symbols more useful for the dynamic linker, the section offset (file interpretation) gives way to a virtual address (memory interpretation) for which the section number is irrelevant.

Although the symbol table values have similar meanings for different object files, the data allows efficient access by the appropriate programs.

Section 2. ELF Program and Dynamic Linking General Information

ELF Program and Dynamic Linking General Information

This section describes the object file information and system actions that create running programs. Some information here applies to all systems; information specific to one processor resides in sections marked accordingly.

Executable and shared object files statically represent programs. To execute such programs, the system uses the files to create dynamic program representations, or process images. As section Virtual Address Space in Chapter 32 of the processor supplement describes, a process image has segments that hold its text, data, stack, and so on. This chapter's major sections discuss the following:

- "Program Header" on page 926. This section complements chapter 32, describing object file structures that relate directly to program execution. The primary data structure, a program header table, locates segment images within the file and contains other information necessary to create the memory image for the program.
- "Program Loading (Processor-Specific)" on page 931. Given an object file, the system must load it into memory for the program to run.
- "Dynamic Linking" on page 931. After the system loads the program it must complete the process image by resolving symbolic references among the object files that compose the process.

NOTE: The processor supplement defines a naming convention for ELF constants that have processor ranges specified. Names such as DT_, PT_, for processor specific extensions, incorporate the name of the

processor: DT_M32_SPECIAL, for example. Pre-existing processor extensions not using this convention will be supported.

Pre-Existing Extensions
DT_JUMP_REL

Program Header

An executable or shared object file's program header table is an array of structures, each describing a segment or other information the system needs to prepare the program for execution. An object file *segment* contains one or more *sections*, as "Segment Contents" on page 929 describes below. Program headers are meaningful only for executable and shared object files. A file specifies its own program header size with the ELF header's **e_phentsize** and **e_phnum** members.

See "ELF Header" on page 899 in chapter 32 for more information.

Program Header

```
typedef struct {
   Elf32_Word   p_type;
   Elf32_Off    p_offset;
   Elf32_Addr   p_vaddr;
   Elf32_Addr   p_paddr;
   Elf32_Word   p_filesz;
   Elf32_Word   p_memsz;
   Elf32_Word   p_flags;
   Elf32_Word   p_align;
} Elf32_Phdr;

typedef struct {
   Elf64_Word   p_type;
   Elf64_Word   p_flags;
   Elf64_Off    p_offset;
   Elf64_Addr   p_vaddr;
   Elf64_Addr   p_paddr;
   Elf64_Xword  p_filesz;
   Elf64_Xword  p_memsz;
   Elf64_Xword  p_align;
} Elf64_Phdr;
```

p_type
This member tells what kind of segment this array element describes or how to interpret the array element's information. Type values and their meanings appear below.

p_offset
This member gives the offset from the beginning of the file at which the first byte of the segment resides.

p_vaddr
This member gives the virtual address at which the first byte of the segment resides in memory.

p_paddr
On systems for which physical addressing is relevant, this member is reserved for the segment's physical address. Because System V ignores physical addressing for application programs, this member has unspecified contents for executable files and shared objects.

p_filesz
This member gives the number of bytes in the file image of the segment; it may be zero.

p_memsz
This member gives the number of bytes in the memory image of the segment; it may be zero.

p_flags
This member gives flags relevant to the segment. Defined flag values appear below.

p_align
As Program Loading describes in this chapter of the processor supplement, loadable process segments must have congruent values for **p_vaddr** and **p_offset**, modulo the page size. This member gives the value to which the segments are aligned in memory and in the file. Values 0 and 1 mean no alignment is required. Otherwise, **p_align** should be a positive, integral power of 2, and **p_vaddr** should equal **p_offset**, modulo **p_align**.

Some entries describe process segments; others give supplementary information and do not contribute to the process image. Segment entries may appear in any order, except as explicitly noted below.

Defined type values follow; other values are reserved for future use.

Segment Types, **p_type table**

Name	Value
PT_NULL	**0**
PT_LOAD	**1**
PT_DYNAMIC	**2**
PT_INTERP	**3**
PT_NOTE	**4**
PT_SHLIB	**5**
PT_PHDR	**6**
PT_LOOS	**0x60000000**
PT_HIOS	**0x6fffffff**
PT_LOPROC	**0x70000000**
PT_HIPROC	**0x7fffffff**

PT_NULL
The array element is unused; other members' values are undefined. This type lets the program header table have ignored entries.

PT_LOAD
The array element specifies a loadable segment, described by **p_filesz** and **p_memsz**. The bytes from the file are mapped to the beginning of the memory segment. If the segment's memory size (**p_memsz**) is larger than the file size (**p_filesz**), the extra bytes are defined to hold the value 0 and to follow the segment's initialized area. The file size may not be larger than the memory size. Loadable segment entries in the program header table appear in ascending order, sorted on the **p_vaddr** member.

PT_DYNAMIC
The array element specifies dynamic linking information. See "Dynamic Section" on page 933 below for more information.

PT_INTERP
The array element specifies the location and size of a null-terminated path name to invoke as an interpreter. This segment type is meaningful only for executable files (though it may occur for shared objects); it may not occur more than once in a file. If it is present, it must precede any loadable segment entry. See "Program Interpreter" on page 931 for more information.

PT_NOTE
The array element specifies the location and size of auxiliary information.

See “Note Section” on page 930 for more information.

PT_SHLIB
This segment type is reserved but has unspecified semantics. Programs that contain an array element of this type do not conform to the ABI.

PT_PHDR
The array element, if present, specifies the location and size of the program header table itself, both in the file and in the memory image of the program. This segment type may not occur more than once in a file. Moreover, it may occur only if the program header table is part of the memory image of the program. If it is present, it must precede any loadable segment entry.

See “Program Interpreter” on page 931 for more information.

PT_LOOS through PT_HIOS
Values in this inclusive range are reserved for operating system-specific semantics.

PT_LOPROC through PT_HIPROC
Values in this inclusive range are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.

NOTE: Unless specifically required elsewhere, all program header segment types are optional. A file’s program header table may contain only those elements relevant to its contents.

Base Address

As Program Loading in this chapter of the processor supplement describes, the virtual addresses in the program headers might not represent the actual virtual addresses of the program’s memory image. Executable files typically contain absolute code. To let the process execute correctly, the segments must reside at the virtual addresses used to build the executable file. On the other hand, shared object segments typically contain position-independent code. This lets a segment’s virtual address change from one process to another, without invalidating execution behavior.

On some platforms, while the system chooses virtual addresses for individual processes, it maintains the *relative* position of one segment to another within any one shared object. Because position-independent code on those platforms uses relative addressing between segments, the difference between virtual addresses in memory must match the difference between virtual addresses in the file. The differences between the virtual address of any segment in memory and the corresponding virtual address in the file is thus a single constant value for any one executable or shared object in a given process. This difference is the *base address*. One use of the base address is to relocate the memory image of the file during dynamic linking.

An executable or shared object file’s base address (on platforms that support the concept) is calculated during execution from three values: the virtual memory load address, the maximum page size, and the lowest virtual address of a program’s loadable segment. To compute the base address, one determines the memory address associated with the lowest **p_vaddr** value for a **PT_LOAD** segment. This address is truncated to the nearest multiple of the maximum page size. The corresponding **p_vaddr** value itself is also truncated to the nearest multiple of the maximum page size. The base address is the difference between the truncated memory address and the truncated **p_vaddr** value.

See this chapter in the processor supplement for more information and examples. Operating System Interface of chapter 32 in the processor supplement contains more information about the virtual address space and page size.

Segment Permissions

A program to be loaded by the system must have at least one loadable segment (although this is not required by the file format). When the system creates loadable segments’ memory images, it gives access permissions as specified in the **p_flags** member.

Segment Flag Bits, p_flags table

Name	Value	Meaning
PF_X	**0x1**	Execute
PF_W	**0x2**	Write
PF_R	**0x4**	Read
PF_MASKOS	**0x0ff00000**	Unspecified
PF_MASKPROC	**0xf0000000**	Unspecified

All bits included in the **PF_MASKOS** mask are reserved for operating system-specific semantics.

All bits included in the **PF_MASKPROC** mask are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.

If a permission bit is 0, that type of access is denied. Actual memory permissions depend on the memory management unit, which may vary from one system to another. Although all flag combinations are valid, the system may grant more access than requested. In no case, however, will a segment have write permission unless it is specified explicitly. The following table shows both the exact flag interpretation and the allowable flag interpretation. ABI-conforming systems may provide either.

Segment Permissions

Flags	Value	Exact	Allowable
none	**0**	All access denied	All access denied
PF_X	**1**	Execute only	Read, execute
PF_W	**2**	Write only	Read, write, execute
PF_W+PF_X	**3**	Write, execute	Read, write, execute
PF_R	**4**	Read only	Read, execute
PF_R+PF_X	**5**	Read, execute	Read, execute
PF_R+PF_W	**6**	Read, write	Read, write, execute
PF_R+PF_W+PF_X	**7**	Read, write, execute	Read, write, execute

For example, typical text segments have read and execute - but not write - permissions. Data segments normally have read, write, and execute permissions.

Segment Contents

An object file segment comprises one or more sections, though this fact is transparent to the program header. Whether the file segment holds one or many sections also is immaterial to program loading. Nonetheless, various data must be present for program execution, dynamic linking, and so on. The diagrams below illustrate segment contents in general terms. The order and membership of sections within a segment may vary; moreover, processor-specific constraints may alter the examples below. See the processor supplement for details.

Text segments contain read-only instructions and data, typically including the following sections described in Chapter 32. Other sections may also reside in loadable segments; these examples are not meant to give complete and exclusive segment contents.

Text Segment

.text
.rodata
.hash
.dynsym
.dynstr
.plt
.rel.got

Data segments contain writable data and instructions, typically including the following sections.

Data Segment

.data
.dynamic
.got
.bss

A **PT_DYNAMIC** program header element points at the **.dynamic** section, explained in “Dynamic Section” on page 933. The **.got** and **.plt** sections also hold information related to position-independent code and dynamic linking. Although the **.plt** appears in a text segment in the previous table, it may reside in a text or a data segment, depending on the processor. See Global Offset Table and Procedure Linkage Table in this section of the processor supplement for details.

As “Sections” on page 906 in Chapter 32 describes, the **.bss** section has the type **SHT_NOBITS**. Although it occupies no space in the file, it contributes to the segment’s memory image. Normally, these uninitialized data reside at the end of the segment, thereby making **p_memsz** larger than **p_filesz** in the associated program header element.

Note Section

Sometimes a vendor or system builder needs to mark an object file with special information that other programs will check for conformance, compatibility, etc. Sections of type **SHT_NOTE** and program header elements of type **PT_NOTE** can be used for this purpose. The note information in sections and program header elements holds a variable amount of entries. In 64-bit objects (files with **e_ident[EI_CLASS]** equal to **ELFCLASS64**), each entry is an array of 8-byte words in the format of the target processor. In 32-bit objects (files with **e_ident[EI_CLASS]** equal to **ELFCLASS32**), each entry is an array of 4-byte words in the format of the target processor. Labels appear below to help explain note information organization, but they are not part of the specification.

Note Information

namesz
descsz
type
name . . .
desc . . .

namesz and name

The first **namesz** bytes in **name** contain a null-terminated character representation of the entry's owner or originator. There is no formal mechanism for avoiding name conflicts. By convention, vendors use their own name, such as **XYZ Computer Company**, as the identifier. If no name is present, **namesz** contains 0. Padding is present, if necessary, to ensure 8 or 4-byte alignment for the descriptor (depending on whether the file is a 64-bit or 32-bit object). Such padding is not included in **namesz**.

descsz and desc

The first **descsz** bytes in **desc** hold the note descriptor. The ABI places no constraints on a descriptor's contents. If no descriptor is present, **descsz** contains 0. Padding is present, if necessary, to ensure 8 or 4-byte alignment for the next note entry (depending on whether the file is a 64-bit or 32-bit object). Such padding is not included in **descsz**.

type This word gives the interpretation of the descriptor. Each originator controls its own types; multiple interpretations of a single type value may exist. Thus, a program must recognize both the name and the type to recognize a descriptor. Types currently must be non-negative. The ABI does not define what descriptors mean.

To illustrate, the following note segment holds two entries.

Example Note Segment

NOTE: The system reserves note information with no name (namesz=0) and with a zero-length name (name[0]='\0') but currently defines no types. All other names must have at least one non-null character.

NOTE: Note information is optional. The presence of note information does not affect a program's ABI conformance, provided the information does not affect the program's execution behavior. Otherwise, the program does not conform to the ABI and has undefined behavior.

Program Loading (Processor-Specific)

NOTE: This section requires processor-specific information. The ABI supplement for the desired processor describes the details.

Dynamic Linking

Program Interpreter

An executable file that participates in dynamic linking shall have one **PT_INTERP** program header element. During **exec**(base operating system), the system retrieves a path name from the **PT_INTERP** segment and creates the initial process image from the interpreter file's segments. That is, instead of using the original executable file's segment images, the system composes a memory image for the interpreter. It then is the interpreter's responsibility to receive control from the system and provide an environment for the application program.

As Process Initialization in Chapter 32 of the processor supplement mentions, the interpreter receives control in one of two ways. First, it may receive a file descriptor to read the executable file, positioned at the beginning. It can use this file descriptor to read and/or map the executable file's segments into memory. Second, depending on the executable file format, the system may load the executable file into memory instead of giving the interpreter an open file descriptor. With the possible exception of the file descriptor, the interpreter's initial process state matches what the executable file would have received. The interpreter itself may not require a second interpreter. An interpreter may be either a shared object or an executable file.

- A shared object (the normal case) is loaded as position-independent, with addresses that may vary from one process to another; the system creates its segments in the dynamic segment area used by

mmap(kernal operating system) and related services [See Virtual Address Space in chapter 32 of the processor supplement]. Consequently, a shared object interpreter typically will not conflict with the original executable file's original segment addresses.

- An executable file may be loaded at fixed addresses; if so, the system creates its segments using the virtual addresses from the program header table. Consequently, an executable file interpreter's virtual addresses may collide with the first executable file; the interpreter is responsible for resolving conflicts.

Dynamic Linker

When building an executable file that uses dynamic linking, the link editor adds a program header element of type **PT_INTERP** to an executable file, telling the system to invoke the dynamic linker as the program interpreter.

NOTE: The locations of the system provided dynamic linkers are processor specific.

Exec(base operating system) and the dynamic linker cooperate to create the process image for the program, which entails the following actions:

- Adding the executable file's memory segments to the process image;
- Adding shared object memory segments to the process image;
- Performing relocations for the executable file and its shared objects;
- Closing the file descriptor that was used to read the executable file, if one was given to the dynamic linker;
- Transferring control to the program, making it look as if the program had received control directly from **exec**(base operating system).

The link editor also constructs various data that assist the dynamic linker for executable and shared object files. As shown above in "Program Header" on page 926, this data resides in loadable segments, making them available during execution. (Once again, recall the exact segment contents are processor-specific. See the processor supplement for complete information).

- A **.dynamic** section with type **SHT_DYNAMIC** holds various data. The structure residing at the beginning of the section holds the addresses of other dynamic linking information.
- The **.hash** section with type **SHT_HASH** holds a symbol hash table.
- The **.got** and **.plt** sections with type **SHT_PROGBITS** hold two separate tables: the global offset table and the procedure linkage table. Chapter 32 discusses how programs use the global offset table for position-independent code. Sections below explain how the dynamic linker uses and changes the tables to create memory images for object files.

Because every ABI-conforming program imports the basic system services from a shared object library [See System Library in Chapter 32], the dynamic linker participates in every ABI-conforming program execution.

As Program Loading explains in the processor supplement, shared objects may occupy virtual memory addresses that are different from the addresses recorded in the file's program header table. The dynamic linker relocates the memory image, updating absolute addresses before the application gains control. Although the absolute address values would be correct if the library were loaded at the addresses specified in the program header table, this normally is not the case.

If the process environment [see **exec**(base operating system)] contains a variable named **LD_BIND_NOW** with a non-null value, the dynamic linker processes all relocations before transferring control to the program. For example, all the following environment entries would specify this behavior.

- **LD_BIND_NOW=1**
- **LD_BIND_NOW=on**
- **LD_BIND_NOW=off**

Otherwise, **LD_BIND_NOW** either does not occur in the environment or has a null value. The dynamic linker is permitted to evaluate procedure linkage table entries lazily, thus avoiding symbol resolution and relocation overhead for functions that are not called. See the Procedure Linkage Table in this chapter of the processor supplement for more information.

Dynamic Section

If an object file participates in dynamic linking, its program header table will have an element of type **PT_DYNAMIC**. This segment contains the **.dynamic** section. A special symbol, **_DYNAMIC**, labels the section, which contains an array of the following structures.

Dynamic Structure

```
typedef struct {
 Elf32_Sword  d_tag;
     union {
       Elf32_Word  d_val;
       Elf32_Addr  d_ptr;
 } d_un;
} Elf32_Dyn;
extern Elf32_Dyn  _DYNAMIC[];
typedef struct {
 Elf64_Sxword  d_tag;
     union {
       Elf64_Xword  d_val;
       Elf64_Addr  d_ptr;
 } d_un;
} Elf64_Dyn;
extern Elf64_Dyn  _DYNAMIC[];
```

For each object with this type, **d_tag** controls the interpretation of **d_un**.

d_val These objects represent integer values with various interpretations.

d_ptr These objects represent program virtual addresses. As mentioned previously, a file's virtual addresses might not match the memory virtual addresses during execution. When interpreting addresses contained in the dynamic structure, the dynamic linker computes actual addresses, based on the original file value and the memory base address.

For consistency, files do not contain relocation entries to correct addresses in the dynamic structure.

To make it simpler for tools to interpret the contents of dynamic section entries, the value of each tag, except for those in two special compatibility ranges, will determine the interpretation of the **d_un** union. A tag whose value is an even number indicates a dynamic section entry that uses **d_ptr**. A tag whose value is an odd number indicates a dynamic section entry that uses **d_val** or that uses neither **d_ptr** nor **d_val**. Tags whose values are less than the special value **DT_ENCODING** and tags whose values fall between **DT_HIOS** and **DT_LOPROC** do not follow these rules.

The following table summarizes the tag requirements for executable and shared object files. If a tag is marked mandatory, the dynamic linking array for an ABI-conforming file must have an entry of that type. Likewise, optional means an entry for the tag may appear but is not required.

Dynamic Array Tags, d_tag

Name	Value	d_un	Executable	Shared Object
DT_NULL	**0**	ignored	mandatory	mandatory
DT_NEEDED	**1**	**d_val**	optional	optional
DT_PLTRELSZ	**2**	**d_val**	optional	optional
DT_PLTGOT	**3**	**d_ptr**	optional	optional

Name	Value	d_un	Executable	Shared Object
DT_HASH	**4**	**d_ptr**	mandatory	mandatory
DT_STRTAB	**5**	**d_ptr**	mandatory	mandatory
DT_SYMTAB	**6**	**d_ptr**	mandatory	mandatory
DT_RELA	**7**	**d_ptr**	mandatory	optional
DT_RELASZ	**8**	**d_val**	mandatory	optional
DT_RELAENT	**9**	**d_val**	mandatory	optional
DT_STRSZ	**10**	**d_val**	mandatory	mandatory
DT_SYMENT	**11**	**d_val**	mandatory	mandatory
DT_INIT	**12**	**d_ptr**	optional	optional
DT_FINI	**13**	**d_ptr**	optional	optional
DT_SONAME	**14**	**d_val**	ignored	optional
DT_RPATH*	**15**	**d_val**	optional	ignored
DT_SYMBOLIC*	**16**	ignored	ignored	optional
DT_REL	**17**	**d_ptr**	mandatory	optional
DT_RELSZ	**18**	**d_val**	mandatory	optional
DT_RELENT	**19**	**d_val**	mandatory	optional
DT_PLTREL	**20**	**d_val**	optional	optional
DT_DEBUG	**21**	**d_ptr**	optional	ignored
DT_TEXTREL*	**22**	ignored	optional	optional
DT_JMPREL	**23**	**d_ptr**	optional	optional
DT_BIND_NOW*	**24**	ignored	optional	optional
DT_INIT_ARRAY	**25**	**d_ptr**	optional	optional
DT_FINI_ARRAY	**26**	**d_ptr**	optional	optional
DT_INIT_ARRAYSZ	**27**	**d_val**	optional	optional
DT_FINI_ARRAYSZ	**28**	**d_val**	optional	optional
DT_RUNPATH	**29**	**d_val**	optional	optional
DT_FLAGS	**30**	**d_val**	optional	optional
DT_ENCODING	**32**	unspecified	unspecified	unspecified
DT_PREINIT_ARRAY	**32**	**d_ptr**	optional	ignored
DT_PREINIT_ARRAYSZ	**33**	**d_val**	optional	ignored
DT_LOOS	**0x6000000D**	unspecified	unspecified	unspecified
DT_HIOS	**0x6ffff000**	unspecified	unspecified	unspecified
DT_LOPROC	**0x70000000**	unspecified	unspecified	unspecified
DT_HIPROC	**0x7fffffff**	unspecified	unspecified	unspecified

* Signifies an entry that is at level 2.

DT_NULL
An entry with a **DT_NULL** tag marks the end of the **_DYNAMIC** array.

DT_NEEDED
This element holds the string table offset of a null-terminated string, giving the name of a needed library. The offset is an index into the table recorded in the **DT_STRTAB** code. See “Shared

Object Dependencies" on page 938 for more information about these names. The dynamic array may contain multiple entries with this type. These entries' relative order is significant, though their relation to entries of other types is not.

DT_PLTRELSZ
This element holds the total size, in bytes, of the relocation entries associated with the procedure linkage table. If an entry of type **DT_JMPREL** is present, a **DT_PLTRELSZ** must accompany it.

DT_PLTGOT
This element holds an address associated with the procedure linkage table and/or the global offset table. See this section in the processor supplement for details.

DT_HASH
This element holds the address of the symbol hash table, described in "Hash Table" on page 940. This hash table refers to the symbol table referenced by the **DT_SYMTAB** element.

DT_STRTAB
This element holds the address of the string table, described in chapter 32. Symbol names, library names, and other strings reside in this table.

DT_SYMTAB
This element holds the address of the symbol table, described in the first part of this chapter, with **Elf32_Sym** entries for the 32-bit class of files and **Elf64_Sym** entries for the 64-bit class of files.

DT_RELA
This element holds the address of a relocation table, described in chapter 32. Entries in the table have explicit addends, such as **Elf32_Rela** for the 32-bit file class or **Elf64_Rela** for the 64-bit file class. An object file may have multiple relocation sections. When building the relocation table for an executable or shared object file, the link editor catenates those sections to form a single table. Although the sections remain independent in the object file, the dynamic linker sees a single table. When the dynamic linker creates the process image for an executable file or adds a shared object to the process image, it reads the relocation table and performs the associated actions. If this element is present, the dynamic structure must also have **DT_RELASZ** and **DT_RELAENT** elements. When relocation is mandatory for a file, either **DT_RELA** or **DT_REL** may occur (both are permitted but not required).

DT_RELASZ
This element holds the total size, in bytes, of the **DT_RELA** relocation table.

DT_RELAENT
This element holds the size, in bytes, of the **DT_RELA** relocation entry.

DT_STRSZ
This element holds the size, in bytes, of the string table.

DT_SYMENT
This element holds the size, in bytes, of a symbol table entry.

DT_INIT
This element holds the address of the initialization function, discussed in "Initialization and Termination Functions" on page 941.

DT_FINI
This element holds the address of the termination function, discussed in "Initialization and Termination Functions" on page 941.

DT_SONAME
This element holds the string table offset of a null-terminated string, giving the name of the shared object. The offset is an index into the table recorded in the **DT_STRTAB** entry. See "Shared Object Dependencies" on page 938 for more information about these names.

DT_RPATH
This element holds the string table offset of a null-terminated search library search path string

discussed in “Shared Object Dependencies” on page 938. The offset is an index into the table recorded in the **DT_STRTAB** entry. This entry is at level 2. Its use has been superseded by DT_RUNPATH.

DT_SYMBOLIC
This element’s presence in a shared object library alters the dynamic linker’s symbol resolution algorithm for references within the library. Instead of starting a symbol search with the executable file, the dynamic linker starts from the shared object itself. If the shared object fails to supply the referenced symbol, the dynamic linker then searches the executable file and other shared objects as usual. This entry is at level 2. Its use has been superseded by the DF_SYMBOLIC flag.

DT_REL
This element is similar to **DT_RELA**, except its table has implicit addends, such as **Elf32_Rel** for the 32-bit file class or **Elf64_Rel** for the 64-bit file class. If this element is present, the dynamic structure must also have **DT_RELSZ** and **DT_RELENT** elements.

DT_RELSZ
This element holds the total size, in bytes, of the **DT_REL** relocation table.

DT_RELENT
This element holds the size, in bytes, of the **DT_REL** relocation entry.

DT_PLTREL
This member specifies the type of relocation entry to which the procedure linkage table refers. The **d_val** member holds **DT_REL** or **DT_RELA**, as appropriate. All relocations in a procedure linkage table must use the same relocation.

DT_DEBUG
This member is used for debugging. Its contents are not specified for the ABI; programs that access this entry are not ABI-conforming.

DT_TEXTREL
This member’s absence signifies that no relocation entry should cause a modification to a non-writable segment, as specified by the segment permissions in the program header table. If this member is present, one or more relocation entries might request modifications to a non-writable segment, and the dynamic linker can prepare accordingly. This entry is at level 2. Its use has been superseded by the DF_TEXTREL flag.

DT_JMPREL
If present, this entry’s **d_ptr** member holds the address of relocation entries associated solely with the procedure linkage table. Separating these relocation entries lets the dynamic linker ignore them during process initialization, if lazy binding is enabled. If this entry is present, the related entries of types **DT_PLTRELSZ** and **DT_PLTREL** must also be present.

DT_BIND_NOW
If present in a shared object or executable, this entry instructs the dynamic linker to process all relocations for the object containing this entry before transferring control to the program. The presence of this entry takes precedence over a directive to use lazy binding for this object when specified through the environment or via **dlopen**(BA_LIB). This entry is at level 2. Its use has been superseded by the DF_BIND_NOW flag.

DT_INIT_ARRAY
This element holds the address of the array of pointers to initialization functions, discussed in “Initialization and Termination Functions” on page 941.

DT_FINI_ARRAY
This element holds the address of the array of pointers to termination functions, discussed in “Initialization and Termination Functions” on page 941.

DT_INIT_ARRAYSZ
This element holds the size in bytes of the array of initialization functions pointed to by the **DT_INIT_ARRAY** entry. If an object has a **DT_INIT_ARRAY** entry, it must also have a **DT_INIT_ARRAYSZ** entry.

DT_FINI_ARRAYSZ
This element holds the size in bytes of the array of termination functions pointed to by the **DT_FINI_ARRAY** entry. If an object has a **DT_FINI_ARRAY** entry, it must also have a **DT_FINI_ARRAYSZ** entry.

DT_RUNPATH
This element holds the string table offset of a null-terminated library search path string discussed in “Shared Object Dependencies” on page 938. The offset is an index into the table recorded in the **DT_STRTAB** entry.

DT_FLAGS
This element holds flag values specific to the object being loaded. Each flag value will have the name **DF**_*flag_name*. Defined values and their meanings are described below. All other values are reserved.

DT_PREINIT_ARRAY
This element holds the address of the array of pointers to pre-initialization functions, discussed in “Initialization and Termination Functions” on page 941. The **DT_PREINIT_ARRAY** table is processed only in an executable file; it is ignored if contained in a shared object.

DT_PREINIT_ARRAYSZ
This element holds the size in bytes of the array of pre-initialization functions pointed to by the **DT_PREINIT_ARRAY** entry. If an object has a **DT_PREINIT_ARRAY** entry, it must also have a **DT_PREINIT_ARRAYSZ** entry. As with **DT_PREINIT_ARRAY**, this entry is ignored if it appears in a shared object.

DT_ENCODING
Values greater than or equal to **DT_ENCODING** and less than **DT_LOOS** follow the rules for the interpretation of the **d_un** union described above.

DT_LOOS through DT_HIOS
Values in this inclusive range are reserved for operating system-specific semantics. All such values follow the rules for the interpretation of the **d_un** union described above.

DT_LOPROC through DT_HIPROC
Values in this inclusive range are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them. All such values follow the rules for the interpretation of the **d_un** union described above.

Except for the **DT_NULL** element at the end of the array, and the relative order of **DT_NEEDED** elements, entries may appear in any order. Tag values not appearing in the table are reserved.

DT_FLAGS values

Name	Value
DF_ORIGIN	**0x1**
DF_SYMBOLIC	**0x2**
DF_TEXTREL	**0x4**
DF_BIND_NOW	**0x8**

DF_ORIGIN
This flag signifies that the object being loaded may make reference to the **$ORIGIN** substitution

string (see “Substitution Sequences” on page 939). The dynamic linker must determine the pathname of the object containing this entry when the object is loaded.

DF_SYMBOLIC
If this flag is set in a shared object library, the dynamic linker’s symbol resolution algorithm for references within the library is changed. Instead of starting a symbol search with the executable file, the dynamic linker starts from the shared object itself. If the shared object fails to supply the referenced symbol, the dynamic linker then searches the executable file and other shared objects as usual.

DF_TEXTREL
If this flag is not set, no relocation entry should cause a modification to a non-writable segment, as specified by the segment permissions in the program header table. If this flag is set, one or more relocation entries might request modifications to a non-writable segment, and the dynamic linker can prepare accordingly.

DF_BIND_NOW
If set in a shared object or executable, this flag instructs the dynamic linker to process all relocations for the object containing this entry before transferring control to the program. The presence of this entry takes precedence over a directive to use lazy binding for this object when specified through the environment or via **dlopen**(BA_LIB).

Shared Object Dependencies

When the link editor processes an archive library, it extracts library members and copies them into the output object file. These statically linked services are available during execution without involving the dynamic linker. Shared objects also provide services, and the dynamic linker must attach the proper shared object files to the process image for execution.

When the dynamic linker creates the memory segments for an object file, the dependencies (recorded in **DT_NEEDED** entries of the dynamic structure) tell what shared objects are needed to supply the program’s services. By repeatedly connecting referenced shared objects and their dependencies, the dynamic linker builds a complete process image. When resolving symbolic references, the dynamic linker examines the symbol tables with a breadth-first search. That is, it first looks at the symbol table of the executable program itself, then at the symbol tables of the **DT_NEEDED** entries (in order), and then at the second level **DT_NEEDED** entries, and so on. Shared object files must be readable by the process; other permissions are not required.

NOTE: Even when a shared object is referenced multiple times in the dependency list, the dynamic linker will connect the object only once to the process.

Names in the dependency list are copies either of the **DT_SONAME** strings or the path names of the shared objects used to build the object file. For example, if the link editor builds an executable file using one shared object with a **DT_SONAME** entry of **lib1** and another shared object library with the path name **/usr/lib/lib2**, the executable file will contain **lib1** and **/usr/lib/lib2** in its dependency list.

If a shared object name has one or more slash (**/**) characters anywhere in the name, such as **/usr/lib/lib2** or **directory/file**, the dynamic linker uses that string directly as the path name. If the name has no slashes, such as **lib1**, three facilities specify shared object path searching.

1. The dynamic array tag **DT_RUNPATH** gives a string that holds a list of directories, separated by colons (**:**). For example, the string **/home/dir/lib:/home/dir2/lib:** tells the dynamic linker to search first the directory **/home/dir/lib**, then **/home/dir2/lib**, and then the current directory to find dependencies.

   The set of directories specified by a given **DT_RUNPATH** entry is used to find only the immediate dependencies of the executable or shared object containing the **DT_RUNPATH** entry. That is, it is used only for those dependencies contained in the **DT_NEEDED** entries of the dynamic structure containing the **DT_RUNPATH** entry, itself. One object’s **DT_RUNPATH** entry does not affect the search for any other object’s dependencies.

2. A variable called **LD_LIBRARY_PATH** in the process environment [see **exec**(base operating system)] may hold a list of directories as above, optionally followed by a semicolon (**;**) and another directory list.

   The following values would be equivalent to the previous example:

   a. **LD_LIBRARY_PATH=/home/dir/usr/lib:/home/dir2/usr/lib:**
   b. **LD_LIBRARY_PATH=/home/dir/usr/lib;/home/dir2/usr/lib:**
   c. **LD_LIBRARY_PATH=/home/dir/usr/lib:/home/dir2/usr/lib:;**

   Although some programs (such as the link editor) treat the lists before and after the semicolon differently, the dynamic linker does not. Nevertheless, the dynamic linker accepts the semicolon notation, with the semantics described previously.

   All **LD_LIBRARY_PATH** directories are searched before those from **DT_RUNPATH**.
3. Finally, if the other two groups of directories fail to locate the desired library, the dynamic linker searches the default directories, **/usr/lib** or such other directories as may be specified by the ABI supplement for a given processor.

When the dynamic linker is searching for shared objects, it is not a fatal error if an ELF file with the wrong attributes is encountered in the search. Instead, the dynamic linker shall exhaust the search of all paths before determining that a matching object could not be found. For this determination, the relevant attributes are contained in the following ELF header fields:

e_ident[EI_DATA], **e_ident[EI_CLASS]**, **e_ident[EI_OSABI]**, **e_ident[EI_ABIVERSION]**, **e_machine**, **e_type**, **e_flags** and **e_version**.

NOTE: For security, the dynamic linker ignores LD_LIBRARY_PATH for set-user and set-group ID programs. It does, however, search DT_RUNPATH directories and the default directories. The same restriction may be applied to processes that have more than minimal privileges on systems with installed extended security mechanisms.

NOTE: A fourth search facility, the dynamic array tag DT_RPATH, has been moved to level 2 in the ABI. It provides a colon-separated list of directories to search. Directories specified by DT_RPATH are searched before directories specified by LD_LIBRARY_PATH.

If both **DT_RPATH** and **DT_RUNPATH** entries appear in a single object's dynamic array, the dynamic linker processes only the **DT_RUNPATH** entry.

Substitution Sequences

Within a string provided by dynamic array entries with the **DT_NEEDED** or **DT_RUNPATH** tags and in pathnames passed as parameters to the **dlopen()** routine, a dollar sign (**$**) introduces a substitution sequence. This sequence consists of the dollar sign immediately followed by either the longest *name* sequence or a name contained within left and right braces (**{**) and (**}**). A name is a sequence of bytes that start with either a letter or an underscore followed by zero or more letters, digits or underscores. If a dollar sign is not immediately followed by a name or a brace-enclosed name, the behavior of the dynamic linker is unspecified.

If the name is **ORIGIN**, then the substitution sequence is replaced by the dynamic linker with the absolute pathname of the directory in which the object containing the substitution sequence originated. Moreover, the pathname will contain no symbolic links or use of **.** or **..** components. Otherwise (when the name is not **ORIGIN**) the behavior of the dynamic linker is unspecified.

When the dynamic linker loads an object that uses **$ORIGIN**, it must calculate the pathname of the directory containing the object. Because this calculation can be computationally expensive, implementations may want to avoid the calculation for objects that do not use **$ORIGIN**. If an object calls **dlopen()** with a string containing **$ORIGIN** and does not use **$ORIGIN** in one if its dynamic array entries, the dynamic linker may not have calculated the pathname for the object until the **dlopen()** actually occurs.

Since the application may have changed its current working directory before the **dlopen()** call, the calculation may not yield the correct result. To avoid this possibility, an object may signal its intention to reference **$ORIGIN** by setting the DF_ORIGIN flag. An implementation may reject an attempt to use **$ORIGIN** within a **dlopen()** call from an object that did not set the **DF_ORIGIN** flag and did not use **$ORIGIN** within its dynamic array.

NOTE: For security, the dynamic linker does not allow use of $ORIGIN substitution sequences for set-user and set-group ID programs. For such sequences that appear within strings specified by DT_RUNPATH dynamic array entries, the specific search path containing the $ORIGIN sequence is ignored (though other search paths in the same string are processed). $ORIGIN sequences within a DT_NEEDED entry or path passed as a parameter to dlopen() are treated as errors. The same restrictions may be applied to processes that have more than minimal privileges on systems with installed extended security mechanisms.

Global Offset Table

NOTE: This section requires processor-specific information. The System V Application Binary Interface supplement for the desired processor describes the details.

Procedure Linkage Table

NOTE: This section requires processor-specific information. The System V Application Binary Interface supplement for the desired processor describes the details.

Hash Table

A hash table of **Elf32_Word** objects supports symbol table access. The same table layout is used for both the 32-bit and 64-bit file class. Labels appear below to help explain the hash table organization, but they are not part of the specification.

Symbol Hash

nbucket
nchain
bucket[0] **. . .** **bucket[nbucket-1]**
chain[0] **. . .** **chain[nchain-1]**

The **bucket** array contains **nbucket** entries, and the **chain** array contains **nchain** entries; indexes start at 0. Both **bucket** and **chain** hold symbol table indexes.

Chain table entries parallel the symbol table. The number of symbol table entries should equal **nchain**; so symbol table indexes also select chain table entries. A hashing function (shown below) accepts a symbol name and returns a value that may be used to compute a **bucket** index.

Consequently, if the hashing function returns the value *x* for some name, **bucket[***x***%nbucket]** gives an index, *y*, into both the symbol table and the chain table.

If the symbol table entry is not the one desired, **chain[***y***]** gives the next symbol table entry with the same hash value.

One can follow the **chain** links until either the selected symbol table entry holds the desired name or the **chain** entry contains the value **STN_UNDEF**.

Hashing Function

```
unsigned long
elf_hash(const unsigned char *name)
{
 unsigned long  h = 0, g;
 while (*name)
 {
   h = (h << 4) + *name++;
   if (g = h & 0xf0000000)
     h ^= g >> 24;
   h &= ~g;
 }
 return h;
}
```

Initialization and Termination Functions

After the dynamic linker has built the process image and performed the relocations, each shared object and the executable file get the opportunity to execute some initialization functions. All shared object initializations happen before the executable file gains control.

Before the initialization functions for any object A is called, the initialization functions for any other objects that object A depends on are called. For these purposes, an object A depends on another object B, if B appears in A's list of needed objects (recorded in the **DT_NEEDED** entries of the dynamic structure). The order of initialization for circular dependencies is undefined.

The initialization of objects occurs by recursing through the needed entries of each object. The initialization functions for an object are invoked after the needed entries for that object have been processed. The order of processing among the entries of a particular list of needed objects is unspecified.

NOTE: Each processor supplement may optionally further restrict the algorithm used to determine the order of initialization. Any such restriction, however, may not conflict with the rules described by this specification.

The following example illustrates two of the possible correct orderings which can be generated for the example NEEDED lists. In this example the *a.out* is dependent on **b**, **d**, and **e**. **b** is dependent on **d** and **f**, while **d** is dependent on **e** and **g**. From this information a dependency graph can be drawn. The above algorithm on initialization will then allow the following specified initialization orderings among others.

Initialization Ordering Example Similarly, shared objects and executable files may have termination functions, which are executed with the **atexit**(base operating system) mechanism after the base process begins its termination sequence. The termination functions for any object A must be called before the termination functions for any other objects that object A depends on. For these purposes, an object A depends on another object B, if B appears in A's list of needed objects (recorded in the **DT_NEEDED** entries of the dynamic structure). The order of termination for circular dependencies is undefined.

Finally, an executable file may have pre-initialization functions. These functions are executed after the dynamic linker has built the process image and performed relocations but before any shared object initialization functions. Pre-initialization functions are not permitted in shared objects.

NOTE: Complete initialization of system libraries may not have occurred when pre-initializations are executed, so some features of the system may not be available to pre-initialization code. In general, use of pre-initialization code can be considered portable only if it has no dependencies on system libraries.

The dynamic linker ensures that it will not execute any initialization, pre-initialization, or termination functions more than once.

Shared objects designate their initialization and termination code in one of two ways. First, they may specify the address of a function to execute via the **DT_INIT** and **DT_FINI** entries in the dynamic structure, described in “Dynamic Section” on page 933 above.

Shared objects may also (or instead) specify the address and size of an array of function pointers. Each element of this array is a pointer to a function to be executed by the dynamic linker. Each array element is the size of a pointer in the programming model followed by the object containing the array. The address of the array of initialization function pointers is specified by the **DT_INIT_ARRAY** entry in the dynamic structure. Similarly, the address of the array of pre-initialization functions is specified by **DT_PREINIT_ARRAY** and the address of the array of termination functions is specified by **DT_FINI_ARRAY**. The size of each array is specified by the **DT_INIT_ARRAYSZ**, **DT_PREINIT_ARRAYSZ**, and **DT_FINI_ARRAYSZ** entries.

The functions whose addresses are contained in the arrays specified by **DT_INIT_ARRAY** and by **DT_PREINIT_ARRAY** are executed by the dynamic linker in the same order in which their addresses appear in the array; those specified by **DT_FINI_ARRAY** are executed in reverse order.

If an object contains both **DT_INIT** and **DT_INIT_ARRAY** entries, the function referenced by the **DT_INIT** entry is processed before those referenced by the **DT_INIT_ARRAY** entry for that object. If an object contains both **DT_FINI** and **DT_FINI_ARRAY** entries, the functions referenced by the **DT_FINI_ARRAY** entry are processed before the one referenced by the **DT_FINI** entry for that object.

NOTE: Although the atexit(base operating system) termination processing normally will be done, it is not guaranteed to have executed upon process death. In particular, the process will not execute the termination processing if it calls _exit [see exit(base operating system)] or if the process dies because it received a signal that it neither caught nor ignored.

The processor supplement for each processor specifies whether the dynamic linker is responsible for calling the executable file’s initialization function or registering the executable file’s termination function with **atexit**(base operating system). Termination functions specified by users via the **atexit**(base operating system) mechanism must be executed before any termination functions of shared objects.

Appendix A. Character Maps

The following tables are a textual representation of the character maps listed in “Code Set Overview” on page 379:

- “ISO Code Sets”
- “IBM Code Sets” on page 961

ISO Code Sets

The following ISO code sets are described:

- “ISO8859–1”
- “ISO8859–2” on page 945
- “ISO8859–5” on page 948
- “ISO8859–6” on page 950
- “ISO8859–7” on page 952
- “ISO8859–8” on page 954
- “ISO8859–9” on page 956
- “ISO8859–15” on page 958

ISO8859–1

Table 12. ISO8859–1 Code set

Symbolic Name	Hex Value
no break space	A0
inverted exclamation mark	A1
cent sign	A2
pound sign	A3
currency sign	A4
yen sign	A5
broken bar	A6
section sign	A7
diaeresis	A8
copyright sign	A9
feminine ordinal indicator	AA
left-pointing double angle quotation mark	AB
not sign	AC
soft hyphen	AD
registered sign	AE
macron	AF
degree sign	B0
plus-minus sign	B1
superscript two	B2
superscript three	B3
acute accent	B4

Table 12. ISO8859–1 Code set (continued)

micro sign	B5
pilcrow sign	B6
middle dot	B7
cedilla	B8
superscript one	B9
masculine ordinal indicator	BA
right-pointing double angle quotation mark	BB
vulgar fraction one quarter	BC
vulgar fraction one half	BD
vulgar fraction three quarters	BE
inverted question mark	BF
latin capital letter A with grave	C0
latin capital letter A with acute	C1
latin capital letter A with circumflex	C2
latin capital letter A with tilde	C3
latin capital letter A with diaeresis	C4
latin capital letter A with ring above	C5
latin capital letter AE	C6
latin capital letter C with cedilla	C7
latin capital letter E with grave	C8
latin capital letter E with acute	C9
latin capital letter E with circumflex	CA
latin capital letter E with diaeresis	CB
latin capital letter I with grave	CC
latin capital letter I with acute	CD
latin capital letter I with circumflex	CE
latin capital letter I with diaeresis	CF
latin capital letter eth	D0
latin capital letter n with tilde	D1
latin capital letter O with grave	D2
latin capital letter O with acute	D3
latin capital letter O with circumflex	D4
latin capital letter O with tilde	D5
latin capital letter O with diaeresis	D6
multiplication sign	D7
latin capital letter O with stroke	D8
latin capital letter U with grave	D9
latin capital letter U with acute	DA
latin capital letter U with circumflex	DB
latin capital letter U with diaeresis	DC
latin capital letter Y with acute	DD

Table 12. ISO8859–1 Code set (continued)

latin capital letter thorn	DE
latin small letter sharp S	DF
latin small letter A with grave	E0
latin small letter A with acute	E1
latin small letter A with circumflex	E2
latin small letter A with tilde	E3
latin small letter A with diaeresis	E4
latin small letter A with ring above	E5
latin small letter AE	E6
latin small letter C with cedilla	E7
latin small letter E with grave	E8
latin small letter E with acute	E9
latin small letter E with circumflex	EA
latin small letter E with diaeresis	EB
latin small letter I with grave	EC
latin small letter I with acute	ED
latin small letter I with circumflex	EE
latin small letter I with diaeresis	EF
latin small letter eth	F0
latin small letter n with tilde	F1
latin small letter O with grave	F2
latin small letter O with acute	F3
latin small letter O with circumflex	F4
latin small letter O with tilde	F5
latin small letter O with diaeresis	F6
division sign	F7
latin small letter O with stroke	F8
latin small letter U with grave	F9
latin small letter U with acute	FA
latin small letter U with circumflex	FB
latin small letter U with diaeresis	FC
latin small letter Y with acute	FD
latin small letter thorn	FE
latin small letter y with diaeresis	FF

ISO8859–2

Table 13. ISO8859–2 Code set

Symbolic Name	**Hex Value**
no break space	A0
latin capital letter A with ogonek	A1

Table 13. ISO8859–2 Code set (continued)

bleve	A2
capital letter L with stroke	A3
currency sign	A4
latin capital letter L with caron	A5
latin capital letter S with acute	A6
section sign	A7
diaeresis	A8
latin capital letter S with caron	A9
latin capital letter S with cedilla	AA
latin capital letter T with caron	AB
latin capital letter Z with acute	AC
soft hyphen	AD
latin capital letter Z with caron	AE
latin capital letter Z with dot above	AF
degree sign	B0
latin small letter A with ogenek	B1
ogenek	B2
latin small letter L with stroke	B3
acute accent	B4
latin small letter L with caron	B5
latin small letter S with acute	B6
caron	B7
cedilla	B8
latin small letter S with caron	B9
latin small letter S with cedilla	BA
latin small letter T with caron	BB
latin small letter Z with acute	BC
double acute accent	BD
latin small letter Z with caron	BE
latin small letter Z with dot above	BF
latin capital letter R with acute	C0
latin capital letter A with acute	C1
latin capital letter A with circumflex	C2
latin capital letter A with breve	C3
latin capital letter A with diaeresis	C4
latin capital letter L with acute	C5
latin capital letter C with acute	C6
latin capital letter C with cedilla	C7
latin capital letter C with caron	C8
latin capital letter E with acute	C9
latin capital letter E with ogonek	CA

Table 13. ISO8859–2 Code set (continued)

latin capital letter E with diaeresis	CB
latin capital letter E with caron	CC
latin capital letter I with acute	CD
latin capital letter I with circumflex	CE
latin capital letter D with caron	CF
latin capital letter D with stroke	D0
latin capital letter N with acute	D1
latin capital letter N with caron	D2
latin capital letter O with acute	D3
latin capital letter O with circumflex	D4
latin capital letter O with double acute	D5
latin capital letter O with diaeresis	D6
multiplication sign	D7
latin capital letter R with caron	D8
latin capital letter U with ring above	D9
latin capital letter U with acute	DA
latin capital letter U with double acute	DB
latin capital letter U with diaeresis	DC
latin capital letter Y with acute	DD
latin capital letter T with cedilla	DE
latin small letter sharp S	DF
latin small letter R with acute	E0
latin small letter A with acute	E1
latin small letter A with circumflex	E2
latin small letter A with breve	E3
latin small letter A with diaeresis	E4
latin small letter L with acute	E5
latin small letter C with acute	E6
latin small letter C with cedilla	E7
latin small letter C with caron	E8
latin small letter E with acute	E9
latin small letter E with ogonek	EA
latin small letter E with diaeresis	EB
latin small letter E with caron	EC
latin small letter I with acute	ED
latin small letter I with circumflex	EE
latin small letter D with caron	EF
latin small letter D with stroke	F0
latin small letter N with acute	F1
latin small letter N with caron	F2
latin small letter O with acute	F3

Table 13. ISO8859–2 Code set (continued)

latin small letter O with circumflex	F4
latin small letter O with double acute	F5
latin small letter O with diaeresis	F6
division sign	F7
latin small letter R with caron	F8
latin small letter U with ring above	F9
latin small letter U with acute	FA
latin small letter Uwith double acute	FB
latin small letter U with diaeresis	FC
latin small letter Y with acute	FD
latin small letter T with cedilla	FE
dot above	FF

ISO8859–5

Table 14. ISO8859–5 Code set

Symbolic Name	**Hex Value**
no break space	A0
cyrillic capital letter io	A1
cyrillic capital letter dje	A2
cyrillic capital letter gje	A3
cyrillic capital letter ukrainian ie	A4
cyrillic capital letter dze	A5
cyrillic capital letter byelorussian-ukranian I	A6
cyrillic capital letter yi	A7
cyrillic capital letter je	A8
cyrillic capital letter lje	A9
cyrillic capital letter nje	AA
cyrillic capital letter tshe	AB
cyrillic capital letter kje	AC
soft hyphen	AD
cyrillic capital letter short U	AE
cyrillic capital letter dzhe	AF
cyrillic capital letter A	B0
cyrillic capital letter be	B1
cyrillic capital letter ve	B2
cyrillic capital letter ghe	B3
cyrillic capital letter de	B4
cyrillic capital letter ie	B5
cyrillic capital letter zhe	B6
cyrillic capital letter ze	B7

Table 14. ISO8859–5 Code set (continued)

cyrillic capital letter I	B8
cyrillic capital letter short I	B9
cyrillic capital letter ka	BA
cyrillic capital letter el	BB
cyrillic capital letter em	BC
cyrillic capital letteren	BD
cyrillic capital letter O	BE
cyrillic capital letter pe	BF
cyrillic capital letter er	C0
cyrillic capital letter es	C1
cyrillic capital letter te	C2
cyrillic capital letter U	C3
cyrillic capital letter ef	C4
cyrillic capital letter ha	C5
cyrillic capital letter tse	C6
cyrillic capital letter che	C7
cyrillic capital letter sha	C8
cyrillic capital letter shcha	C9
cyrillic capital letter hard sign	CA
cyrillic capital letter yeru	CB
cyrillic capital letter soft sign	CC
cyrillic capital letter E	CD
cyrillic capital letter tu	CE
cyrillic capital letter ya	CF
cyrillic small letter A	D0
cyrillic small letter be	D1
cyrillic small letter ve	D2
cyrillic small letter ghe	D3
cyrillic small letter de	D4
cyrillic small letter ie	D5
cyrillic small letter zhe	D6
cyrillic small letter ze	D7
cyrillic small letter I	D8
cyrillic small letter short I	D9
cyrillic small letter ka	DA
cyrillic small letter el	DB
cyrillic small letter em	DC
cyrillic small letter en	DD
cyrillic small letter O	DE
cyrillic small letter pe	DF
cyrillic small letter er	E0

Table 14. ISO8859–5 Code set (continued)

cyrillic small letter es	E1
cyrillic small letter te	E2
cyrillic small letter U	E3
cyrillic small letter ef	E4
cyrillic small letter ha	E5
cyrillic small letter tse	E6
cyrillic small letter che	E7
cyrillic small letter sha	E8
cyrillic small letter shcha	E9
cyrillic small letter hard sign	EA
cyrillic small letter yeru	EB
cyrillic small letter soft sign	EC
cyrillic small letter E	ED
cyrillic small letter yu	EE
cyrillic small letter ta	EF
numero sign	F0
cyrillic small letter io	F1
cyrillic small letter dje	F2
cyrillic small letter gje	F3
cyrillic small letter ukranian ie	F4
cyrillic small letter dze	F5
cyrillic small letter byelorussian-ukranian I	F6
cyrillic small letter yi	F7
cyrillic small letter je	F8
cyrillic small letter lje	F9
cyrillic small letter nje	FA
cyrillic small letter tshe	FB
cyrillic small letter kje	FC
selection sign	FD
cyrillic small letter short U	FE
cyrillic small letter dzhe	FF

ISO8859–6

Table 15. ISO8859–6

Symbolic Name	**Hex Value**
no-break space	A0
currency sign	A4
Arabic comma	AC
soft hyphen	AD
Arabic semicolon	BB

Table 15. ISO8859–6 (continued)

Arabic question mark	BF
Arabic letter hamza	C1
Arabic letter alef with madda above	C2
Arabic letter alef with hamza above	C3
Arabic letter waw with hamza above	C4
Arabic letter alef with hamza below	C5
Arabic letter yeh with hamza above	C6
Arabic letter alef	C7
Arabic letter beh	C8
Arabic letter teh marbuta	C9
Arabic letter teh	CA
Arabic letter theh	CB
Arabic letter jeem	CC
Arabic letter hah	CD
Arabic letter khah	CE
Arabic letter dal	CF
Arabic letter thal	D0
Arabic letter reh	D1
Arabic letter zain	D2
Arabic letter seen	D3
Arabic letter sheen	D4
Arabic letter sad	D5
Arabic letter dad	D6
Arabic letter tah	D7
Arabic letter zah	D8
Arabic letter ain	D9
Arabic letter ghain	DA
Arabic letter tatweel	E0
Arabic letter feh	E1
Arabic letter qaf	E2
Arabic letter kaf	E3
Arabic letter lam	E4
Arabic letter meem	E5
Arabic letter noon	E6
Arabic letter heh	E7
Arabic letter waw	E8
Arabic letter alef maksura	E9
Arabic letter yeh	EA
Arabic letter fathatan	EB
Arabic letter dammatan	EC
Arabic letter kasratan	ED

Table 15. ISO8859–6 (continued)

Arabic letter fatha	EE
Arabic letter damma	EF
Arabic letter kasra	F0
Arabic letter shadda	F1
Arabic letter sukun	F2

ISO8859–7

Table 16. ISO8859–7 Code set

Symbolic Name	Hex Value
no break space	A0
left single quotation mark	A1
right single quotation mark	A2
puond sign	A3
euro sign	A4
broken bar	A6
section sign	A7
diaeresis	A8
copyright sign	A9
left-pointing double angle quotation mark	AB
not sign	AC
soft hyphen	AD
horizontal bar	AF
degree sign	B0
plus-minus sign	B1
superscript two	B2
superscript three	B3
greek tonos	B4
greek dialytika tonos	B5
greek capital letter alpha with tonos	B6
middle dot	B7
greek capital letter epsilon with tonos	B8
greek capital letter eta with tonos	B9
greek capital letter iota with tonos	BA
right-pointing double angle quotation mark	BB
greek capital letter omicron with tonos	BC
vulgar fraction one half	BD
greek capital letter upsilon with tonos	BE
greek capital letter omega with tonos	BF
greek small letter iota with dialytika and tonos	C0
greek capital letter alpha	C1

Table 16. ISO8859–7 Code set (continued)

greek capital letter beta	C2
greek capital letter gamma	C3
greek capital letter delta	C4
greek capital letter epsilon	C5
greek capital letter zeta	C6
greek capital letter eta	C7
greek capital letter theta	C8
greek capital letter iota	C9
greek capital letter kappa	CA
greek capital letter lamda	CB
greek capital letter mu	CC
greek capital letter nu	CD
greek capital letter xi	CE
greek capital letter omicron	CF
greek capital letter pi	D0
greek capital letter rho	D1
greek capital letter sigma	D3
greek capital letter tau	D4
greek capital letter upsilon	D5
greek capital letter phi	D6
greek capital letter chi	D7
greek capital letter psi	D8
greek capital letter omega	D9
greek capital letter iota with dialytika	DA
greek capital letter upsilon with dialytika	DB
greek small letter alpha with tonos	DC
greek small letter epsilon with tonos	DD
greek small letter eta with tonos	DE
greek small letter iota with tonos	DF
greek small letter upsilon with dialytika and tonos	E0
greek small letter alpha	E1
greek small letter beta	E2
greek small letter gamma	E3
greek small letter delta	E4
greek small letter epsilon	E5
greek small letter zeta	E6
greek small letter eta	E7
greek small letter theta	E8
greek small letter iota	E9
greek small letter kappa	EA
greek small letter lamda	EB

Table 16. ISO8859–7 Code set (continued)

greek small letter mu	EC
greek small letter nu	ED
greek small letter xi	EE
greek small letter omicron	EF
greek small letter pi	F0
greek small letter rho	F1
greek small letter final sigma	F2
greek small letter sigma	F3
greek small letter tau	F4
greek small letter upsilon	F5
greek small letter phi	F6
greek small letter chi	F7
greek small letter psi	F8
greek small letter omega	F9
greek small letter iota with dialytika	FA
greek small letter upsilon with dialytika	FB
greek small letter omicron with tonos	FC
greek small letter upsilon with tonos	FD
greek small letter omega with tonos	FE

ISO8859–8

Table 17. ISO8859–8 Code set

Symbolic Name	**Hex Value**
no-break space	A0
cent sign	A2
pound sign	A3
currency sign	A4
yen sign	A5
broken bar	A6
section sign	A7
diaeresis	A8
copyright sign	A9
multiplication sign	AA
left-pointing double angle quotation mark	AB
not sign	AC
soft hyphen	AD
registered sign	AE
overline	AF
degree sign	B0
plus-minus sign	B1

Table 17. ISO8859–8 Code set (continued)

superscript two	B2
superscript three	B3
acute accent	B4
micro sign	B5
pilcrow sign	B6
middle dot	B7
cedilla	B8
superscript one	B9
division sign	BA
right-pointing double angle quotation mark	BB
vulgar fraction one quarter	BC
vulgar fraction one half	BD
vilgar fraction three quarters	BE
double low line	DF
hebrew letter alef	EO
hebrew letter bet	E1
hebrew letter gimel	E2
hebrew letter dalet	E3
hebrew letter he	E4
hebrew letter vav	E5
hebrew letter zayin	E6
hebrew letter het	E7
hebrew letter tet	E8
hebrew letter yod	E9
hebrew letter final kaf	EA
hebrew letter kaf	EB
hebrew letter lamed	EC
hebrew letter final mem	ED
hebrew letter mem	EE
hebrew letter final nun	EF
hebrew letter nun	F0
hebrew letter samekh	F1
hebrew letter ayin	F2
hebrew letter final pe	F3
hebrew letter pe	F4
hebrew letter final tsadi	F5
hebrew letter tsadi	F6
hebrew letter qof	F7
hebrew letter resh	F8
hebrew letter shin	F9
hebrew letter tav	FA

ISO8859–9

Table 18. ISO8859–9 Code set

Symbolic Name	Hex Value
no-break space	A0
inverted exclamation mark	A1
cent sign	A2
pound sign	A3
currency sign	A4
yen sign	A5
broken bar	A6
section sign	A78
diaeresis	A8
copyright sign	A9
feminine ordinal indicator	AA
left-pointing double quotation mark	AB
not sign	AC
sofy hyphen	AD
registered sign	AE
macron	AF
degree sign	B0
plus-minus sign	B1
superscript two	B2
superscript three	B3
acute accent	B4
micro sign	B5
pilcrow sign	B6
middle dot	B7
cedilla	B8
superscript one	B9
masculine ordinal indicator	BA
right pointing double angle quotation mark	BB
vulgar fraction one quarter	BC
vulgar fraction one half	BD
vulgar fraction three quarters	BE
inverted question mark	BF
latin capital letter A with grave	C0
latin capital letter A with acute	C1
latin capital letter A with circumflex	C2
latin capital letter A with tilde	C3
latin capital letter A with diaeresis	C4

Table 18. ISO8859–9 Code set (continued)

latin capital letter A with ring above	C5
latin capital letter AE	C6
latin capital letter C with cedilla	C7
latin capital letter E with grave	C8
latin capital letter E with acute	C9
latin capital letter E with circumflex	CA
latin capital letter E with diaeresis	CB
latin capital letter I with grave	CC
latin capital letter I with acute	CD
latin capital letter I with circumflex	CE
latin capital letter I with diaeresis	CF
latin capital letter G with breve	D0
latin capital letter N with tilde	D1
latin capital letter O with grave	D2
latin capital letter O with acute	D3
latin capital letter O with circumflex	D4
latin capital letter O with tilde	D5
latin capital letter O with diaeresis	D6
multiplication sign	D7
latin capital letter O with stroke	D8
latin capital letter U with grave	D9
latin capital letter U with acute	DA
latin capital letter U with circumflex	DB
latin capital letter U with diaeresis	DC
latin capital letter I with dot above	DD
latin capital letter S with cedilla	DE
latin small letter sharp S	DF
latin small letter A with grave	E0
latin small letter A with acute	E1
latin small letter A with circumflex	E2
latin small letter A with tilde	E3
latin small letter A with diaeresis	E4
latin small letter A with ring above	E5
latin small letter AE	E6
latin small letter C with cedilla	E7
latin small letter E with grave	E8
latin small letter E with acute	E9
latin small letter E with circumflex	EA
latin small letter E with diseresis	EB
latin small letter I with grave	EC
latin small letter I with acute	ED

Table 18. ISO8859–9 Code set (continued)

latin small letter I with circumflex	EE
latin small letter I with diaeresis	EF
latin small letter G with breve	F0
latin small letter N with tilde	F1
latin small letter O with grave	F2
latin small letter O with acute	F3
latin small letter O with circumflex	F4
latin small letter O with tilde	F5
latin small letter O with diaeresis	F6
division sign	F7
latin small letter O with stroke	F8
latin small letter U with grave	F9
latin small letter U with acute	FA
latin small letter U with circumflex	FB
latin small letter U with diaeresis	FC
latin small letter dotless I	FD
latin small letter S with cedilla	FE
latin small letter Y with diaeresis	FF

ISO8859–15

Table 19. ISO8859–1 Code set

Symbolic Name	**Hex Value**
no-break space	A0
inverted exclamation mark	A1
cent sign	A2
pound sign	A3
euro sign	A4
yen sign	A5
latin capital letter S with caron	A6
section sign	A7
letin small letter S with caron	A8
copyright sign	A9
feminine ordinal indicator	AA
left-pointing double angle quotation mark	AB
not sign	AC
soft hyphen	AD
registered sign	AE
macron	AF
degree sign	B0
plus-minus sign	B1

Table 19. ISO8859–1 Code set (continued)

superscript two	B2
superscript three	B3
latin capital letter Z with caron	B4
micro sign	B5
pilcrow sign	B6
middle dot	B7
latin small letter Z with caron	B8
superscript one	B9
masculine ordinal indicator	BA
right-pointing bouble angle quotation marks	BB
latin capital ligature oe	BC
latin small ligature oe	BD
latin capital letter Y with diaeresis	BE
inverted question mark	BF
latin capital letter A with grave	C0
latin capital letter A with acute	C1
latin capital letter A with circumflex	C2
latin capital letter A with tilde	C3
latin capital letter A with diaeresis	C4
latin capital letter A with ring above	C5
latin capital letter AE	C6
latin capital letter C with cedilla	C7
latin capital letter E with grave	C8
latin capital letter E with acute	C9
latin capital letter W with circumflex	CA
latin capital letter E with diaeresis	CB
latin capital letter I with grave	CC
latin capital letter I with acute	CD
latin capital letter I with circumflex	CE
latin capital letter I with diaeresis	CF
latin capital letter eth	D0
latin capital letter N with tilde	D1
latin capital letter O with grave	D2
latin capital letter O with acute	D3
latin capital letter O with circumflex	D4
latin capital letter O with tilde	D5
latin capital letter O with diaeresis	D6
multiplication sign	D7
latin capital letter O with stroke	D8
latin capital letter U with grave	D9
latin capital letter U with acute	DA

Table 19. ISO8859–1 Code set (continued)

latin capital letter U with circumflex	DB
latin capital letter U with diaeresis	DC
latin capital letter Y with acute	DD
latin capital letter thorn	DE
latin small letter sharp S	DF
latin small letter A with grave	EO
latin small letter A with acute	E1
latin small letter A with circumflex	E2
latin small letter A with tilde	E3
latin small letter A with diaeresis	E4
latin small letter A with ring above	E5
latin small letter AE	E6
latin small letter C with cedilla	E7
latin small letter E with grave	E8
latin small letter E with acute	E9
latin small letter E with circumflex	EA
latin small letter E with diaeresis	EB
latin small letter I with grave	EC
latin small letter I with acute	ED
latin small letter I with circumflex	EE
latin small letter I with diaeresis	EF
latin small letter eth	F0
latin small letter N with tilde	F1
latin small letter O with grave	F2
latin small letter O with acute	F3
latin small letter O with circumflex	F4
latin small letter O with tilde	F5
latin small letter O with diaeresis	F6
division sign	F7
latin small letter O with stroke	F8
latin small letter U with grave	F9
latin small letter U with acute	FA
latin small letter U with circumflex	FB
latin small letter U with diaeresis	FC
latin small letter Y with acute	FD
latin small letter thorn	FE
latin small letter Y with diaeresis	FF

IBM Code Sets

The following IBM PC code sets are described:

IBM-850

Table 20. IBM—850 Code set

Symbolic Name	Hex Value
delete	7F
latin capital letter C with cedilla	80
latin small letter U with diaeresis	81
latin small letter E with acute	82
latin small letter A with circumflex	83
latin small letter A with diaeresis	84
latin small letter A with grave diaeresis	85
latin small letter A with ring above	86
latin small letter C with cedilla	87
latin small letter E with circumflex	88
latin small letter E with diaeresis	89
latin small letter E with grave	8A
latin small letter I with diaeresis	8B
latin small letter I with circumflex	8C
latin small letter I with grave	8D
latin capital letter A with diaeresis	8E
latin capital letter A with ring above	8F
latin capital letter E with acute	90
latin small letter AE	91
latin capital letter AE	92
latin small letter O with circumflex	93
latin small letter O with diaeresis	94
latin small letter O with grave	95
latin small letter U with circumflex	96
latin small letter U with grave	97
latin small letter Y with diaeresis	98
latin capital letter O with diaeresis	99
latin capital letter U with diaeresis	9A

Table 20. IBM—850 Code set (continued)

latin small letter O with stroke	9B
pound sign	9C
latin capital letter O with stroke	9D
multiplication sign	9E
latin small letter F with hook	9F
latin small letter A with acute	A0
latin small letter I with acute	A1
latin small letter O with acute	A2
latin small letter U with acute	A3
latin small letter N with tilde	A4
latin capital letter N with tilde	A5
feminie ordinal indicator	A6
masculine ordinal indicator	A7
inverted question mark	A8
registered sign	A9
not sign	AA
vulgar fraction one half	AB
vulgar fraction one quarter	AC
inverted exclamation mark	AD
left-pointing double angle quotation mark	AE
right-pointing double angle quotation mark	AF
light shade	B0
medium shade	B1
dark shade	B2
box drawings light vertical	B3
box drawings light vertical and left	B4
latin capital letter A with acute	B5
latin capital letter A with circumflex	B6
latin capital letter A with grave	B7
copyright sign	B8
box drawings double vertical and left	B9
box drawings double vartical	BA
box drawings double down and left	BB
box drawings double up and left	BC
cent sign	BD
yen sign	BE
box drawings light down and left	BF
box drawings light up and right	C0
box drawings light up abd horizontal	C1
box drawings light down and horizontal	C2
box drawings light vertical and right	C3

Table 20. IBM—850 Code set (continued)

box drawings light horizontal	C4
box drawings light vertical and horizontal	C5
latin small letter A with tilde	C6
latin capital letter A with tilde	C7
box drawings double up and right	C8
box drawings double down and right	C9
box drawings double up and horizontal	CA
box drawings double down and horizontal	CB
box drawings double vertical and right	CC
box drawings double horizontal	CD
box drawings double vertical and horizontal	CE
currency sign	CF
latin small letter eth	D0
latin capital letter eth	D1
latin capital letter E with circumflex	D2
latin capital letter E with diaeresis	D3
latin capital letter E with grave	D4
latin small letter dotless I	D5
latin capital letter I with acute	D6
latin capital letter I with circumflex	D7
latin capital letter I with diaeresis	D8
box drawings light up and left	D9
box drawings right down and right	DA
full block	DB
lower half block	DC
broken bar	DD
latin capital letter I with grave	DE
upper half block	DF
latin capital letter O with acute	E0
latin small letter sharp S	E1
latin capital letter O with circumflex	E2
latin capital letter O with grave	E3
latin small letter O with tilde	E4
latin capital letter O with tilde	E5
micro sign	E6
latin small capital letter thorn	E7
latin capital letter thorn	E8
latin capital letter U with acute	E9
latin capital letter U with circumflex	EA
latin capital letter U with grave	EB
latin small letter U with acute	EC

Table 20. IBM—850 Code set (continued)

latin capital letter Y with acute	ED
macron	EE
acute accent	EF
soft hyphen	F0
plus-minus sign	F1
double low line	F2
vulgar fraction three quarters	F3
pilcrow sign	F4
section sign	F5
division sign	F6
cedilla	F7
degree sign	F8
diaeresis	F9
middle dot	FA
superscript one	FB
superscript three	FC
superscript two	FD
black square	FE
no-break space	FF

IBM-856

Table 21. IBM–856 Code set

Symbolic Name	**Hex Value**
hebrew letter alef	80
hebrew letter bet	81
hebrew letter gimel	82
hebrew letter dalet	83
hebrew letter he	84
hebrew letter vav	85
hebrew letter zayin	86
hebrew letter het	87
hebrew letter tet	88
hebrew letter yod	89
hebrew letter final kaf	8A
hebrew letter kaf	8B
hebrew letter lamed	8C
hebrew letter final mem	8D
hebrew letter mem	8E
hebrew letter final nun	8F
hebrew letter nun	90

Table 21. IBM–856 Code set (continued)

hebrew letter samekh	91
hebrew letter ayin	92
hebrew letter final pe	93
hebrew letter pe	94
hebrew letter final tsadi	95
hebrew letter tsadi	96
hebrew letter qof	97
hebrew letter resh	98
hebrew letter shin	99
hebrew letter tav	9A
pound sign	9C
multiplication sign	9E
registered sign	A9
not sign	AA
vulgar fraction one half	AB
vulgar fraction one quarter	AC
left pointing double angle quotation mark	AE
right pointing double angle quotation mark	AF
light shade	B0
medium shade	B1
dark shade	B2
box drawings light vertical	B3
box drawings light vertical and left	B4
copyright sign	B8
box drawings double vertival and left	B9
box drawings double vertical	BA
box drawings double down and left	BB
box drawings double up and left	BC
cent sign	BD
yen sign	BE
box drawings light down and left	BF
box drawings light up and right	C0
box drawings light up and horizontal	C1
box drawings light down and horizontal	C2
box drawings light vertical and right	C3
box drawings light horizontal	C4
box drawings light vertical and horizontal	C5
box drawings double up and right	C8
box drawings double down and right	C9
box drawings double up and horizontal	CA
box drawings double down and horizontal	CB

Table 21. IBM–856 Code set (continued)

box drawings double vertical and right	CC
box drawings double horizontal	CD
box drawings double vertical and horizontal	CE
currency sign	CF
box drawings light up and left	D9
box drawings light down and right	DA
full block	DB
lower half block	DC
broken bar	DD
upper half block	DF
micro sign	E6
overline	EE
acute accent	EF
soft hyphen	F0
plus-minus sign	F1
double low line	F2
vulgar fraction three quarters	F3
pilcrow sign	F4
section sign	F5
division sign	F6
cedilla	F7
degree sign	F8
diaeresis	F9
middle dot	FA
superscript one	FB
superscript three	FC
superscript two	FD
black square	FE
no-break space	FF

IBM-921

Table 22. IBM–921 Code set

Symbolic Name	**Hex Value**
no-break space	A0
right double quotation mark	A1
cent sign	A2
pound sign	A3
euro sign	A4
double low-9 quotation mark	A5
broken bar	A6

Table 22. IBM–921 Code set (continued)

section sign	A7
latin capital letter O with stroke	A8
copyright sign	A9
latin capital letter R with cedilla	AA
left-pointing double angle quotation mark	AB
not sign	AC
soft hyphen	AD
registered sign	AE
latin capital letter AE	AF
degree sign	B0
plus-minus sign	B1
superscript two	B2
superscript three	B3
left double quotation mark	B4
micro sign	B5
pilcrow sign	B6
middle dot	B7
latin small letter O with stroke	B8
superscript one	B9
latin small letter R with cedilla	BA
right-pointing double angle quotation mark	BB
vulgar fraction one quarter	BC
vulgar fraction one half	BD
vulgar fraction three quarters	BE
latin small letter AE	BF
latin capital letter A with ogonek	C0
latin capital letter I with ogonek	C1
latin capital letter A with macron	C2
latin capital letter C with acute	C3
latin capital letter A with diaeresis	C4
latin capital letter A with ring above	C5
latin capital letter E with ogonek	C6
latin capital letter E with macron	C7
latin capital letter C with caron	C8
latin capital letter E with acute	C9
latin capital letter Z with acute	CA
latin capital letter E with dot above	CB
latin capital letter G with cedilla	CC
latin capital letter K with cedilla	CD
latin capital letter I with macron	CE
latin capital letter L with cedilla	CF

Table 22. IBM–921 Code set (continued)

latin capital letter S with caron	D0
latin capital letter N with acute	D1
latin capital letter N with cedilla	D2
latin capital letter O with acute	D3
latin capital letter O with macron	D4
latin capital letter O with tilde	D5
latin capital letter O with diaeresis	D6
multiplication sign	D7
latin capital letter U with ogonek	D8
latin capital letter L with stroke	D9
latin capital letter S with acute	DA
latin capital letter U with macron	DB
latin capital letter U with diaeresis	DC
latin capital letter Z with dot above	DD
latin capital letter Z with caron	DE
latin small letter sharp S	DF
latin small letter A with ogonek	E0
latin small letter I with ogonek	E1
latin small letter A with macron	E2
latin small letter C with acute	E3
latin small letter A with diaeresis	E4
latin small letter A with ring above	E5
latin small letter E with ogonek	E6
latin small letter E with macron	E7
latin small letter C with caron	E8
latin small letter E with acute	E9
latin small letter Z with acute	EA
latin small letter E with dot above	EB
latin small letter G with cedilla	EC
latin small letter K with cedilla	ED
latin small letter I with macron	EE
latin small letter L with cedilla	EF
latin small letter S with caron	F0
latin small letter N with acute	F1
latin small letter N with cedilla	F2
latin small letter O with acute	F3
latin small letter O with macron	F4
latin small letter O with tilde	F5
latin small letter O with diaeresis	F6
division sign	F7
latin small letter U with ogonek	F8

Table 22. IBM–921 Code set (continued)

latin small letter L with stroke	F9
latin small letter S with acute	FA
latin small letter U with macron	FB
latin small letter U with diaeresis	FC
latin small letter Z with dot above	FD
latin small letter Z with caron	FE
right single quotation mark	FF

IBM-922

Table 23. IBM–922 Code set

Symbolic Name	Hex Value
no break space	A0
inverted excamation mark	A1
cent sign	A2
pound sign	A3
euro sign	A4
yenb sign	A5
broken bar	A6
section sign	A7
diaeresis	A8
copyright sign	A9
feminine ordinal indicator	AA
left-pointing double angle quotation mark	AB
not sign	AC
soft hyphen	AD
registered sign	AE
macron	AF
degree sign	B0
plus-minus sign	B1
superscript two	B2
superscript three	B3
acute accent	B4
micro sign	B5
pilcrow sign	B6
middle dot	B7
cedilla	B8
superscript one	B9
masculine ordinal indicator	BA
right-pointing double angle quotation mark	BB
vulgar fraction one quarter	BC

Table 23. IBM–922 Code set (continued)

vulgar fraction one half	BD
vulgar fraction three quarters	BE
inverted question mark	BF
latin capital letter A with grave	C0
latin capital letter A with acute	C1
latin capital letter A with circumflex	C2
latin capital letter A with tilde	C3
latin capital letter A with diaeresis	C4
latin capital letter A with ring above	C5
latin capital letter AE	C6
latin capital letter C with cedilla	C7
latin capital letter E with grave	C8
latin capital letter E with acute	C9
latin capital letter E with circumflex	CA
latin capital letter E with diaeresis	CB
latin capital letter I with grave	CC
latin capital letter I with acute	CD
latin capital letter I with circumflex	CE
latin capital letter I with diaeresis	CF
latin capital letter S with caron	D0
latin capital letter N with tilde	D1
latin capital letter O with grave	D2
latin capital letter O with acute	D3
latin capital letter O with circumflex	D4
latin capital letter O with tilde	D5
latin capital letter O with diaeresis	D6
multiplication sign	D7
latin capital letter O with stroke	D8
latin capital letter U with grave	D9
latin capital letter U with acute	DA
latin capital letter U with circuflex	DB
latin capital letter U with diaeresis	DC
latin capital letter Y with acute	DD
latin capital letter Z with caron	DE
latin small letter sharp S	DF
latin small letter A with grave	E0
latin small letter A with acute	E1
latin small letter A with circumflex	E2
latin small letter A with tilde	E3
latin small letter A with diaeresis	E4
latin small letter A with ring above	E5

Table 23. IBM–922 Code set (continued)

latin small letter AE	E6
latin small letter C with cedilla	E7
latin small letter E with grave	E8
latin small letter E with acute	E9
latin small letter E with circumflex	EA
latin small letter E with diaeresis	EB
latin small letter I with grave	EC
latin small letter I with acute	ED
latin small letter I with curcumflex	EE
latin small letter I with diaeresis	EF
latin small letter S with caron	F0
latin small letter N with tilde	F1
latin small letter O with grave	F2
latin small letter O with acute	F3
latin small letter O with circumflex	F4
latin small letter O with tilde	F5
latin small letter O with diaeresis	F6
division sign	F7
latin small letter O with stroke	F8
latin small letter U with grave	F9
latin small letter U with acute	FA
latin small letter U with circumflex	FB
latin small letter U with diaeresis	FC
latin small letter Y with acute	FD
latin small letter Z with caron	FE
latin small letter Y with diaeresis	FF

IBM-1046

Table 24. IBM–1046 Code set

Symbolic Name	**Hex Value**
arabic letter alef with hamza below final form	80
multiplication sign	81
division sign	82
arabic letter seen first part of final form	83
arabic letter sheen first part of final form	84
arabic letter sad first part of final form	85
arabic letter dadfirst part of final form	86
arabic tatweel with fathatan above	87
full block	89
box drawings light vertical	8A

Table 24. IBM–1046 Code set (continued)

box drawings light horizontal	8B
box drawings light down and left	8C
box drawings light down and right	8D
box drawings light up and right	8E
box drawings light up and left	8F
arabic damma medial form	90
arabic kasra medial form	91
arabic shadda medial form	92
arabic sukun medial form	93
arabic fatha medial form	94
arabic letter yeh with hamza above final form	95
arabic letter alef maksura final form	96
arabic letter yeh initial form	97
arabic letter yeh final form	98
arabic letter ghain final form	99
arabic letter ghain initial form	9A
arabic letter ghain medial form	9B
arabic ligature lam with alef with madda above final form	9C
arabic ligature lam with alef with hamza above final form	9D
arabic ligature lam with alef with hamza below final form	9E
arabic ligature lam with alef final form	9f
no-break space	A0
arabic letter alef with madda above after lam	A1
arabic letter alef with hamza above after lam	A2
arabic letter alef with hamza below after lam	A3
currency sign	A4
arabic letter alef after lam	A5
arabic letter yeh with hamza above initial form	A6
arabic letter beh with initial form	A7
arabic letter teh with initial form	A8
arabic letter theh with initial form	A9
arabic letter jeem with initial form	AA
arabic letter hah with initial form	AB
arabic comma	AC
soft hyphen	AD
arabic letter khan initial form	AE
arabic letter seen initial form	AF
arabic-indic digit zero	B0
arabic-indic digit one	B1
arabic-indic digit two	B2
arabic-indic digit three	B3

Table 24. IBM–1046 Code set (continued)

arabic-indic digit four	B4
arabic-indic digit five	B5
arabic-indic digit six	B6
arabic-indic digit seven	B7
arabic-indic digit eight	B8
arabic-indic digit nine	B9
arabic letter sheen initial form	BA
arabic semicolon	BB
arabic letter sad initial form	BC
arabic letter dad initial form	BD
arabic letter ain initial form	BE
arabic question mark	BF
arabic letter ain initial form	C0
arabic letter hamza	C1
arabic letter alef with madda above	C2
arabic letter alef with hamza above	C3
arabic letter waw with hamza above	C4
arabic letter alef with hamza below	C5
arabic letter yeh with hamza above	C6
arabic letter alef	C7
arabic letter beh	C8
arabic letter teh marbuta	C9
arabic letter teh	CA
arabic letter theh	CB
arabic letter jeem	CC
arabic letter hah	CD
arabic letter khah	CE
arabic letter dal	CF
arabic letter thal	D0
arabic letter reh	D1
arabic letter zain	D2
arabic letter seen	D3
arabic letter sheen	D4
arabic letter sad	D5
arabic letter dad	D6
arabic letter tah	D7
arabic letter zah	D8
arabic letter ain	D9
arabic letter ghain	DA
arabic letter ain medial form	DB
arabic letter alef with madda above final form	DC

Table 24. IBM–1046 Code set (continued)

arabic letter alef with hamza above final form	DD
arabic letter alef with final form	DE
arabic letter feh initial form	DF
arabic tatweel	E0
arabic letter feh	E1
arabic letter qaf	E2
arabic letter kaf	E3
arabic letter lam	E4
arabic letter meem	E5
arabic letter noon	E6
arabic letter heh	E7
arabic letter waw	E8
arabic letter alef maksura	E9
arabic letter yeh	EA
arabic fathatan	EB
arabic dammatan	EC
arabic kasratan	ED
arabic fatha	EE
arabic damma	EF
arabic kasra	F0
arabic shadda	F1
arabic sukun	F2
arabic letter qar initial form	F3
arabic letter kaf initial form	F4
arabic letter lam initial form	F5
arabic kasseh	F6
arabic ligature lam with alef with madda above isolated form	F7
arabic ligature lam with alef with hamza above isolated form	F8
arabic ligature lam with alef with madda below isolated form	F9
arabic ligature lam with alef isolated form	FA
arabic letter meem initial form	FB
arabic letter noon initial form	FC
arabic letter heh initial form	FD
arabic letter heh final form	FE
euro sign	FF

IBM-1124

Table 25. IBM–1124 Code set

Symbolic Name	Hex Value
no-break space	A0

Table 25. IBM–1124 Code set (continued)

cyrillic capital letter io	A1
cyrillic capital letter dje	A2
cyrillic capital letter ghe with upturn	A3
cyrillic capital letter ukranian ie	A4
cyrillic capital letter dze	A5
cyrillic capital letter byelorussian-ukranian i	A6
cyrillic capital letter yi	A7
cyrillic capital letter je	A8
cyrillic capital letter lje	A9
cyrillic capital letter nje	AA
cyrillic capital letter tshe	AB
cyrillic capital letter kje	AC
soft hyphen	AD
cyrillic capital letter short U	AE
cyrillic capital letter dzhe	AF
cyrillic capital letter A	B0
cyrillic capital letter be	B1
cyrillic capital letter ve	B2
cyrillic capital letter ghe	B3
cyrillic capital letter de	B4
cyrillic capital letter ie	B5
cyrillic capital letter zhe	B6
cyrillic capital letter ze	B7
cyrillic capital letter I	B8
cyrillic capital letter short I	B9
cyrillic capital letter ka	BA
cyrillic capital letter el	BB
cyrillic capital letter em	BC
cyrillic capital letter en	BD
cyrillic capital letter O	BE
cyrillic capital letter pe	BF
cyrillic capital letter er	C0
cyrillic capital letter es	C1
cyrillic capital letter te	C2
cyrillic capital letter U	C3
cyrillic capital letter ef	C4
cyrillic capital letter ha	C5
cyrillic capital letter tse	C6
cyrillic capital letter che	C7
cyrillic capital letter sha	C8
cyrillic capital letter shcha	C9

Table 25. IBM–1124 Code set (continued)

cyrillic capital letter hard sign	CA
cyrillic capital letter yeru	CB
cyrillic capital letter soft sign	CC
cyrillic capital letter E	CD
cyrillic capital letter yu	CE
cyrillic capital letter ya	CF
cyrillic small letter A	D0
cyrillic small letter be	D1
cyrillic small letter ve	D2
cyrillic small letter ghe	D3
cyrillic small letter de	D4
cyrillic small letter ie	D5
cyrillic small letter zhe	D6
cyrillic small letter ze	D7
cyrillic small letter I	D8
cyrillic small letter short I	D9
cyrillic small letter ka	DA
cyrillic small letter el	DB
cyrillic small letter em	DC
cyrillic small letter en	DD
cyrillic small letter O	DE
cyrillic small letter pe	DF
cyrillic small letter er	E0
cyrillic small letter es	E1
cyrillic small letter te	E2
cyrillic small letter u	E3
cyrillic small letter ef	E4
cyrillic small letter ha	E5
cyrillic small letter tse	E6
cyrillic small letter che	E7
cyrillic small letter sha	E8
cyrillic small letter shcha	E9
cyrillic small letter hard sign	EA
cyrillic small letter yeru	EB
cyrillic small letter soft sign	EC
cyrillic small letter E	ED
cyrillic small letter yu	EE
cyrillic small letter ya	EF
numero sign	F0
cyrillic small letter io	F1
cyrillic small letter dje	F2

Table 25. IBM–1124 Code set (continued)

cyrillic small letter ghe with upturn	F3
cyrillic small letter ukrainian ie	F4
cyrillic small letter dze	F5
cyrillic small letter byelorussian-ukrainian	F6
cyrillic small letter yi	F7
cyrillic small letter je	F8
cyrillic small letter lje	F9
cyrillic small letter nje	FA
cyrillic small letter tshe	FB
cyrillic small letter kje	FC
section sign	FD
cyrillic small letter short u	FE
cyrillic small letter dzhe	FF

IBM-1129

Table 26. IBM–1129 Code set

Symbolic Name	**Hex Value**
no-break space	A0
inverted exclamation mark	A1
cent sign	A2
pound sign	A3
euro sign	A4
yen sign	A5
broken bar	A6
section sign	A7
latin small ligature OE	A8
copyright sign	A9
feminine ordinal indicator	AA
left pointing double angle quotation mark	AB
not sign	AC
soft hyphen	AD
registered sign	AE
macron	AF
degree sign	B0
plus-minus sign	B1
superscript two	B2
superscript three	B3
latin capital Y with diaeresis	B4
micro sign	B5
pilcrow sign	B6

Table 26. IBM–1129 Code set (continued)

middle dot	B7
latin capitol ligature OE	B8
superscript one	B9
masculine ordinal indicator	BA
right pointing double angle quotation mark	BB
vulgar fraction one quarter	BC
vulgar fraction one half	BD
vulgar fraction three quarters	BE
inverted question mark	BF
latin capitol letter A with grave	C0
latin capitol letter A with acute	C1
latin capitol letter A with circumflex	C2
latin capitol letter A with breve	C3
latin capitol letter A with diaeresis	C4
latin capitol letter A with ring above	C5
latin capitol letter AE	C6
latin capitol letter C with cedilla	C7
latin capitol letter E with grave	C8
latin capitol letter E with acute	C9
latin capitol letter E with circumflex	CA
latin capitol letter E with diaeresis	CB
combining grave accent	CC
latin capitol letter I with acute	CD
latin capitol letter I with circumflex	CE
latin capitol letter I with diaeresis	CF
latin capitol letter D with stroke	D0
latin capitol letter N with tilde	D1
combining hook above	D2
latin capitol letter O with acute	D3
latin capitol letter O with circumflex	D4
latin capitol letter O with horn	D5
latin capitol letter O with diaeresis	D6
multiplication sign	D7
latin capitol letter O with stroke	D8
latin capitol letter U with grave	D9
latin capitol letter U with acute	DA
latin capitol letter U with circuflex	DB
latin capitol letter U with diaeresis	DC
latin capitol letter U with horn	DD
combining tilde	DE
latin small letter sharp S	DF

Table 26. IBM–1129 Code set (continued)

latin small letter A with grave	E0
latin small letter A with acute	E1
latin small letter A with circumflex	E2
latin small letter A with breve	E3
latin small letter A with diaeresis	E4
latin small letter A with ring above	E5
latin small letter AE	E6
latin small letter C with cedilla	E7
latin small letter E with grave	E8
latin small letter E with acute	E9
latin small letter E with circumflex	EA
latin small letter E with diaeresis	EB
combining acute accent	EC
latin small letter I with acute	ED
latin small letter I with circumflex	EE
latin small letter I with diaeresis	EF
latin small letter D with stroke	F0
latin small letter N with tilde	F1
combining dot below	F2
latin small letter O with acute	F3
latin small letter O with circumflex	F4
latin small letter O with horn	F5
latin small letter O with diaeresis	F6
division sign	F7
latin small letter O with stroke	F8
latin small letter U with grave	F9
latin small letter U with acute	FA
latin small letter U with circumflex	FB
latin small letter U with diaeresis	FC
latin small letter U with horn	FD
dong sign	FE
latin small letter Y with diaeresis	FF

TIS-620

Table 27. TIS–620 Code set

Symbolic Name	**Hex Value**
thai character ko kai	A1
thai character kho khai	A2
thai character kho khuat	A3
thai character kho khwai	A4

Table 27. TIS–620 Code set (continued)

thai character kho khon	A5
thai character kho rakhang	A6
thai character ngo ngu	A7
thai character cho chan	A8
thai character cho ching	A9
thai character cho chang	AA
thai character so so	AB
thai character cho choe	AC
thai character yo ying	AD
thai character do chada	AE
thai character to patak	AF
thai character tho than	B0
thai character tho nangmontho	B1
thai character tho phuthao	B2
thai character no nen	B3
thai character do dek	B4
thai character to tao	B5
thai character tho thung	B6
thai character tho thahan	B7
thai character tho thong	B8
thai character no nu	B9
thai character bo baimai	BA
thai character po pla	BB
thai character pho phung	BC
thai character fo fa	BD
thai character pho phan	BE
thai character fo fan	BF
thai character pho samphao	C0
thai character mo ma	C1
thai character yo yak	C2
thai character ro rua	C3
thai character ru	C4
thai character lo ling	C5
thai character lu	C6
thai character wo waen	C7
thai character so sala	C8
thai character so rusi	C9
thai character so sua	CA
thai character ho hip	CB
thai character lo chula	CC
thai character o ang	CD

Table 27. TIS–620 Code set (continued)

thai character ho nokhuk	CE
thai character paiyannoi	CF
thai character sara a	D0
thai character mai han-akat	D1
thai character sara aa	D2
thai character sara am	D3
thai character sara i	D4
thai character sara ii	D5
thai character sara ue	D6
thai character sara uee	D7
thai character sara u	D8
thai character uu	D9
thai character phinthu	DA
thai currency symbol baht	DF
thai character sara e	E0
thai character sara ae	E1
thai character sara O	E2
thai character sara ai maimuan	E3
thai character sara ai maimalai	E4
thai character lakkhangyao	E5
thai character maiyamok	E6
thai character maitaikhu	E7
thai character mai ek	E8
thai character mai tho	E9
thai character mai tri	EA
thai character mai chattawa	EB
thai character thanthakhat	EC
thai character nikhahit	ED
thai character yamakkan	EE
thai character fongman	EF
thai digit zero	F0
thai digit one	F1
thai digit two	F2
thai digit three	F3
thai digit four	F4
thai digit five	F5
thai digit six	F6
thai digit seven	F7
thai digit eight	F8
thai digit nine	F9
that character angkhankhu	FA

Table 27. TIS–620 Code set (continued)

thai character khomut	FB

Appendix B. 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.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:
IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

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.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

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.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and

cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Index

Special Characters

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Readers' Comments — We'd Like to Hear from You

AIX 5L Version 5.1
General Programming Concepts:
Writing and Debugging Programs

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

PLACE
POSTAGE
STAMP
HERE

IBM Corporation
Publications Department
Internal Zip 9561
11400 Burnet Road
Austin, TX
78758-3493

Fold and Tape | **Please do not staple** | Fold and Tape

Cut or Fold
Along Line

Printed in U.S.A